National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Conformance Testing of a DQDB Protocol with SMURPH Observers

by ©

Nyan Tjing Lo

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-82010-1

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Nyan Tjing Lo

TITLE OF THESIS: Conformance Testing of a DQDB Protocol with SMURPH Observers

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Signed: NyanLo

Permanent Address:
35, Rahim Kajan 2
Taman Tun Dr. Ismial
60000 KL, Malaysia

Date: Dec 30, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Conformance Testing of a DQDB Protocol with SMURPH Observers** submitted by **Nyan Tjing Lo** in partial fulfillment of the requirements for the degree of Master of Science.

. . . .
co-supervisor: P. Rudnicki

. . . .
co-supervisor: P. Gburzyński

. . . .
examiner: E. El-Mallah

. . . .
external: W. Joerg (Electrical Engineering)

Date: Dec 28, 1992

Abstract

As the number of various implementations of different protocols increases, validation of these protocols becomes an important issue. Protocol validation can be fulfilled by two main types of reachability analysis algorithms: *exhaustive* state search and *random* state search. *Exhaustive* search attempts to explore all the possible system states during the execution of the system. *Random* state exploration analyzes only a subset of all the reachable system states. Although preferable, *exhaustive* search is usually impractical when applied to complex systems which admit astronomical number of states. *Random* search, currently a subject of active research, can never yield the same confidence as an *exhaustive* search, but it is always feasible. In this thesis, we studied the feasibility of protocol validation by *random* state exploration technique in `smurph`—a software system for modeling protocols. Our interest is in MAC (medium access control) protocols, for which validation is difficult because they involve time-dependent properties. We conducted a case study of a MAC protocol called Distributed Queue Dual Bus, as described in its proposed standard draft. The protocol has been implemented in `smurph` and its conformance with the service specifications of the draft was tested. Several `smurph` tools including observers were used in the process and testing was extensive. Our case study concluded that `smurph` is a useful tool for protocol prototyping and conformance testing of the prototype.

Acknowledgements

I wish to thank my supervisors, Pawel Gburzyǹski and Piotr Rudnicki for their guidance, patience, and precious criticisms throughout this research. In addition, I would like to thank the members of my examining committees, Ehab El-Mallah and Werner Joerg. Their comments and suggestions were valuable in improving this work.

A heartfelt thank you to Manoj Jain for proofreading, encouragement and advice during the writing of this thesis. Finally, I am grateful to my parents, my brother and my sister for their support.

Abbreviations

ACF	Access Control Field
AI	Activity-Interpreter
AU	Access Unit
BAsize	Buffer Allocation Size
BEtag	Beginning-End Tag
BOM	Beginning of Message
BOT	Beginning of Transmission
CD	CountDown Counter
COM	Continuation of Message
CPDU	Common Protocol Data Unit
DA	Destination Address
DMPDU	Derived MAC Protocol Data Unit
DQDB	Distributed Queue Dual Bus protocol
DQSM	Distributed Queue State Machine
EOM	End Of Message
ETU	Experiment Time Unit
IMPDU	Initial MAC Protocol Data Unit
ITU	Indivisible Time Unit
IUT	Implementation Under Test
LAN	Local Area Network
LLC	Logical Link Control
MAC	Medium Access Control
MAN	Metropolitan Area Network
Mbps	Megabits per second
MCF	MAC Convergence Function
MCP	MAC Convergence Protocol

MID	Message IDentifier
MSDU	MAC Service Data Unit
PA	Pre-Arbitrated
PCO	Point Of Control and Observation
PI	Protocol Identification
PL	PAD Length
PSR	Previous Segment Received
QA	Queued Arbitrated
QAF	Queued Arbitrated Function Block
QAP	Queued Arbitrated Portion
QAR	Queued Arbitrated Receive
QOS	Quality Of Service
QPSX	Queued Packet and Synchronous Switch
REQ	REQuest Counter
RQSM	Request Queue State Machine
RSM	Reassembly State Machine
SA	Source Address
SSM	Single Segment Message
SUT	System Under Test
TCP	Test Coordination Procedure
TMPDU	Test Management Protocol Data Unit
TSN	Transmit Sequence Number
VCI	Virtual Channel Identifier

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Protocol Simulation 2

1.3 Thesis objective and Outline 4

2 Conformance Testing **6**

2.1 Introduction 6

2.2 Types of Testing 7

2.2.1 Duologue Matrix Theory 8

2.2.2 Exhaustive Reachability Analysis 9

2.2.3 Random State Exploration 10

2.3 Conformance Testing with Observers 11

2.3.1 Local Test Method 11

2.3.2 Distributed Test Method 13

2.3.3 Observers in Real and Simulated Environments 15

3 The Distributed Queue Dual Bus Protocol **17**

3.1 Introduction 17

3.2 The Dual Bus Topology 18
3.3 Slot Formation 20
3.4 Distributed Queueing Algorithm 21
    3.4.1 A Single Priority Level 22
    3.4.2 Three levels of Priority 23
3.5 The MAC services to the LLC 26
    3.5.1 Creation of IMPDU 27
    3.5.2 Creation of DMPDU 29
    3.5.3 Reassembly process 32

4 SMURPH — An Overview **34**
4.1 Introduction 34
4.2 A Simple Modeling Example 35
4.3 The Modeling Environment 38
    4.3.1 Time 39
    4.3.2 Network Topology 39
    4.3.3 Traffic Patterns 40
    4.3.4 Processes 40
    4.3.5 Process Synchronization 42

5 The Implementation of DQDB in SMURPH **44**
5.1 Introduction 44
5.2 Station Types 45
5.3 Network Configuration 46
5.4 Traffic definition 48

5.5 The Protocol Code 49
  5.5.1 Slotter Processes 52
  5.5.2 MAC Convergence Function process 53
  5.5.3 Queued Arbitrated Portion process 56
  5.5.4 Monitor process 56
  5.5.5 Distributed Queue State Machine 58
  5.5.6 Request Queue State Machine 59
  5.5.7 Transmitter process 59
  5.5.8 Queued Arbitrated Receive process 60
  5.5.9 Reassembly State Machine 61
5.6 Summary 61

6 Conformance Testing of DQDB by Observers 64

6.1 Introduction 64
6.2 SMURPH Observers 66
6.3 DQDB observers 70
  6.3.1 Slot Delay 71
  6.3.2 Priority Traffic Patterns 74
  6.3.3 Slot Request 75
  6.3.4 Validating Distributed Queued State Machine 75
  6.3.5 Validating MAC Convergence Function 77
  6.3.6 Validating Queued Arbitrated Portion process 78
  6.3.7 Validating Queued Arbitrated Received process 78
  6.3.8 Request Queued State Machine Observer 79
6.4 Summary 80

7 Conclusions and Directions for Future Research 81

Bibliography 85

A The Implementation of DQDB 89

A.1 Unsynchronized Slot Generator 89

A.2 MAC Convergence Function Block 89

A.3 Queued Arbitrated Portion 94

A.4 Monitor 96

A.5 Distributed Queue State Machine 98

A.6 Request Queue State Machine 99

A.7 Transmitter 100

A.8 Queued Arbitrated Receive Block 100

A.9 Reassembly State Machine 101

B The Implementation of DQDB Observers 103

B.1 Slot Delay 103

B.2 Queued Arbitrated Portion Observer 104

B.3 DQSM Observer 106

B.4 MCF Observer 108

B.5 RQSM Observer 110

B.6 Queued Arbitrated Receive Observer 112

B.7 Transmitter Observer 113

List of Figures

2.1 The Local Method 12
2.2 The Distributed Method 14

3.1 The Dual Bus Topology 19
3.2 The Slot format of DQDB MAN 20
3.3 The DQSM transition diagram 24
3.4 The interaction between MCF Block and QAF Block 27
3.5 The Initial MAC Protocol Data Unit 28
3.6 The structure of CPDU Header and Trailer 29
3.7 The structure of MAC Convergence Protocol Header 29
3.8 The Derived MAC Protocol Date Unit 30
3.9 The QA Segment 32

4.1 The transition diagram of the `Monitor` process 38
4.2 The life cycle of a `smurph` process 42

5.1 The interaction between station processes 51
5.2 The safety bits 53
5.3 The MCF transition diagram 55
5.4 The QAS transition diagram 57

Chapter 1

Introduction

1.1 Motivation

As the use of computer networks increases rapidly, protocol implementations have become the center of attention in the research community. These protocol implementations must be compatible and robust in order to achieve the goal of Open Systems Interconnection (OSI). However, many protocol specifications have been written in natural language, which is often ambiguous. These specifications can lead to divergent interpretations, and result in incompatibilities in communication systems. This problem can be resolved with formal specification and validation of the communication protocols.

Formal specification defines the precise description of the protocol entities, and thus diminishes some of the ambiguities, incompleteness and inconsistency caused by natural language specifications. Several formal specification languages that seem to prevail are: finite state machines, formal grammars, Petri nets, algebraic calculi, high-level programming languages, abstract data types, and temporal logic [13, 31]. Although such languages can express real-time aspects such as absence of deadlocks and livelocks, completeness, and stability, they are inadequate to address the aspects of protocol data unit encoding [31].

The term ***protocol validation*** is used as a generic term for investigating the correctness of a protocol design [24]. Since even the same formal specification may lead to different implementations, the necessity for a mechanism to validate protocol specifications or implementations is no longer in doubt. The two well-known approaches to protocol validation are:

- exhaustive state search
- random state exploration (e.g. simulation)

The exhaustive approach checks all possible system states that can take place during the execution of the system. Since exploring all the states of a complex system can be infeasible, these methods are usually applied to simplified systems. Random state exploration, on the other hand, analyzes only a fraction of all the reachable states of a system, but it is always feasible. Experience shows that they reveal the most probable behaviour of the systems [38] with no guarantee that the implementation under test is faultless.

1.2 Protocol Simulation

The design of protocols is usually based on a systematic procedure which includes: requirement analysis and definition, development of service specification, implementation of protocol entity and implementation testing. The two crucial assumptions about this process are [13]:

- It is an iterative process. The design is unlikely to be correct the first time it is programmed, and very likely it will not be quite correct the second or third time around either.
- Each time a design phase is completed, the designers should be convinced that it is error-free. A manual walk-through of the code can reveal the biggest blunders, but cannot be expected to reveal the subtle ones. Almost by definition, the designers will overlook the unexpected cases that can cause errors.

Since error correction is expensive in the later phases of the development process, validation should be conducted as early as possible. A legitimate approach of validating a protocol is to implement it and test the actual implementation. However, this approach can be costly if flaws found in the implementation require the system to be redesigned. One way of minimizing the cost is to implement a protocol specification in a simulated environment and test that all the service properties are met before a hardware prototype is built. Additional advantages of the simulated environment are: the designers have complete control over the simulation, the implementations can be modified easily, and tests which are impossible in a physical world (e.g. instantaneous knowledge of events) can be admitted. The adequacy of the simulated environment, however, is outside the scope of this thesis.

Several specification languages namely Estelle [4, 5], LOTOS [32], and SDL [23] have been developed by CCITT and ISO for describing protocols and services. However, these languages fail to reproduce many of the activities occurring at the physical layers. To overcome this limitation, we used `smurph`, a System for Modeling Unslotted Real-time Phenomena [11]. To derive a `smurph` specification, the users are required to manually translate a protocol specification into an extended finite state machine model coded in C++. In fact, the process can be iterative. The derived specification can be perceived as a refinement of the original protocol specification. Running this specification is the same as executing the protocol. Given that `smurph` has the capability of reflecting all the relevant communication phenomena occurring at lower layers, can it be used to perform conformance testing on a standardized protocol? The support of self-checking tools called *observers* seems promising. Why observers? This is because violation of global properties that involve combined actions of multiple processes cannot be detected by simply executing the specification. We need tools which have the ability to monitor all the state transitions that occur while the simulated system is running without affecting its performance. Indeed, the system is unaware that it is being observed.

1.3 Thesis objective and Outline

The purpose of this work is to investigate the feasibility and usefulness of conformance testing in `smurph`. This investigation is based on a case study of the Distributed Queue Dual Bus protocol, as described in its proposed standard draft [14]. There are three significant reasons for selecting the protocol as a testbed for conformance testing. First, a detailed outline of the proposed protocol is available. Second, the complexity of the protocol is suitable for demonstrating the difficulties of developing and implementing observers to detect possible implementation flaws. Finally, it is the first experiment in `smurph` of implementing a standardized protocol with all the possible details.

Substantial effort had been put into manually translating the DQDB specification into a `smurph` specification. One of the difficulties was in maintaining the semantics of the protocol while writing it as `smurph` processes. Another problem was in designing protocol processes such that their state transitions facilitate observers to monitor the required behaviour of processes. Nevertheless, the design process required a number of changes to the initial implementation of the DQDB protocol. Service properties such as absence of deadlocks or livelocks were not tested explicitly since `smurph` detects them during the execution of the DQDB specification. A number of service specifications were validated by passive `smurph` observers, which were constructed according to the models extracted manually from the draft. No errors were reported by the observers in the investigation.

Chapter 2 reviews the techniques of conformance testing. In Chapter 3, the DQDB protocol is presented. It includes the distributed queueing algorithm with one and three priorities. Chapter 4 gives a brief overview of the `smurph` modeling environment. In Chapter 5, the implementation details of the DQDB protocol in `smurph` are delineated. Chapter 6 describes `smurph` observers and how they are used to validate the DQDB protocol. The last Chapter summarizes the results.

Chapter 2

Conformance Testing

2.1 Introduction

The purpose of conformance testing is to determine whether the behaviour of an implementation of a protocol satisfies its formal specification. The given protocol implementation is usually called the *Implementation Under Test* (IUT). The testing schemes attempt to assess the *static* and *dynamic* aspects of the IUT. Static aspects of testing focus on the validity of implementation choices regarding options and services offered. Dynamic aspects of testing, on the other hand, concentrate on events sequencing and the behaviour of the IUT.

Testing is done by stimulating the IUT with a series of test inputs and monitoring the responses generated by the IUT. The observed outputs are compared with the prospective outcomes given in the specification. A result of conformance testing is called a *verdict*. A verdict can be positive or negative. A positive verdict implies that the IUT complies with the protocol specification; whereas a negative verdict indicates that the IUT is incorrectly implemented. Section 2.2 introduces the essentials of *functional* and *structural* testing and some of the existing techniques. A brief overview of the methodology and framework of conformance testing with observers is presented in section 2.3.

2.2 Types of Testing

A *test suite* is a collection of separate test cases that may be executed alone or together. An effective test suite has a good chance of detecting potential errors of an IUT. Generally, test suites derived for an IUT dictates the types of testing (*functional* or *structural*) to be carried out.

Functional testing ignores the internal details of an IUT and focuses on the following aspects [13]:

- Verifying that a given implementation realizes all functions of the original specification, over the full range of parameter values. *Basic interconnection tests* and *capability tests* fall into this class. Basic interconnection tests determine whether an IUT is capable of communicating with other IUTs. Capability tests, on the other hand, check whether the abilities of an IUT are consistent with the static conformance requirements.
- Verifying that a given implementation can properly reject erroneous inputs in a way that is consistent with the original specification. The test suites selected for this type of testing are usually a subset of the infinite set of possible test suites.

Structural testing is relatively more complex than functional testing. It emphasizes the internal structure of the system. In structural testing, the three assumptions listed below are required [13]:

- The IUT models a deterministic finite state machine with a known maximum number of states and with a known input and output vocabulary.
- The IUT produces a response to an input signal within a known, finite amount of time.
- The states and the transitions of the IUT form a strongly connected graph: every state in the graph is reachable from every other state in the machine via one or more transitions.

Both types of test can be performed by various automated validation techniques: *duologue matrix theory*, *exhaustive reachability analysis* and *random state exploration.*

2.2.1 Duologue Matrix Theory

The duologue matrix theory technique has been first introduced by West and Zafiropulo [37, 40] in 1978. This approach studies the interactions between two communicating finite state machines. The two machines are assumed to begin at an initial state. Within a machine, a path starting from the initial state, linking several intermediate states before returning to the initial state is referred to as a unilogue. A duologue exists when the two machines communicate with each other and each machine follows an unilogue path. Validation is done by executing the complete set of duologues derived from all the possible combination of unilogues; the observed results are analyzed. Two types of error: deadlocks and unspecified receptions can be detected by this technique. The major drawback of this method is that it can only apply to a protocol involving two entities. Nevertheless, its application to the CCITT X.21 recommendation demonstrated that the technique is feasible and could find errors in practical protocols [39].

2.2.2 Exhaustive Reachability Analysis

An exhaustive reachability analysis technique attempts to explore all the reachable system states of a protocol from a given initial state. The simplest algorithm is presented below [36]:

1. For all possible combinations of a state *i* and an input *j*, perform the next three steps.
2. Reset the IUT to the initial state, and then apply the appropriate inputs to the IUT so that it arrives at state *i*.
3. Apply input *j* and verify any output received by comparing the output required by the specification.
4. Verify that this final state matches the one required by the specification.

Although this technique tests all the reachable IUT states, it requires the IUT to restart at the initial state after every test. This requirement can be avoided by deriving either multiple ***transition tours*** [28] or ***Unique Input and Output*** (UIO) [29] sequences from the graph of reachable IUT states. A more desired approach is to perform a tree-like exploration of the graph using either a breadth first search or a depth first search. If a breadth first search is employed, all state transitions from a given initial state are analyzed in the order that they are reached. Since all states are reached through one of the shortest paths from the initial state, errors are easy to interpret. If a depth first search is used, all transitions from a given initial state are examined in the inverse order that they are reached (using a stack to hold those states that have been generated but not yet fully explored).

A tree-like exploration method has the following drawbacks. First, it does not reflect the actual behaviour of the system. Second, given that exploring a graph require backtracking, all the previously reached states have to be saved. Storing these states can be expensive. Lastly, the number of states of a complex system to be analyzed can be enormous. Several techniques have been developed to deal with this problem, see [16, 18, 35].

2.2.3 Random State Exploration

The random state exploration technique is simple. It can be perceived as a reduced form of exhaustive search. Instead of systematically exploring all of the state transitions, only one random transition from the current state is analyzed. Problems such as backtracking, search space explosion and state storing are avoided, since the system is executed in continuous sequences rather than in a series of disjoint sequences. In fact, the random state search can apply to all protocols as it is independent of the size and complexity of the system being modeled. This method has two major disadvantages [13]:

- There is no well-defined termination. An exhaustive state exploration of the reachable state space terminates when there are no further states to explore. In random state exploration, there is no way of detecting when all reachable states of the system have been visited.
- It can not guarantee the implementation is error free.

These deficiencies are of lesser importance if exhaustive search is infeasible or impossible. Furthermore, this approach has evinced to be effective and reliable [38] since most errors can be found by exploring only a subset of the total state space. In this case study, we adopted the random state exploration technique since it reflects the most probable behaviour of a system, and is more flexible (not requiring any test sequence generation) when applied to complex systems.

2.3 Conformance Testing with Observers

The concept of observer, introduced in [1], can be used to supplement the validation techniques discussed in the previous section. An observer can be either *passive* or *active.* Active observers interact with the IUT by controlling the inputs and monitoring the outputs (for example, by performing exhaustive search); whereas passive observers only monitor the IUT without interfering.

Both types of observers can be classified into two categories: *local* or *global.* Local observers control and observe the behaviour of a layer n entities within the *System Under Test* (SUT). Global observers, on the other hand, control and observe the behaviour of layer n entities in a system remotely from the SUT. Global observers can be divided into three subtypes: *distributed*, *coordinated* and *remote* [17, 25, 26, 30]. Local observers are not suited for verifying global properties since they can not detect errors requiring global information. Moreover, they are unable to reveal conceivable errors in lower layers. Global observers not only take into consideration the propagation delay between entities, but also can detect hardware errors and potential errors in the lower layer. Detailed description of local and global observers can be found in [33, 34]. Section 2.3.1 presents a framework of local test method followed by the local observers. The concept of distributed test method used by global observers is described in section 2.3.2. Section 2.3.3 discusses testing with observers in real and simulated environment.

2.3.1 Local Test Method

With the local test method, interfaces existing above and below the IUT are exposed. These interfaces, which are called the *Points of Control and Observations* (PCOs),

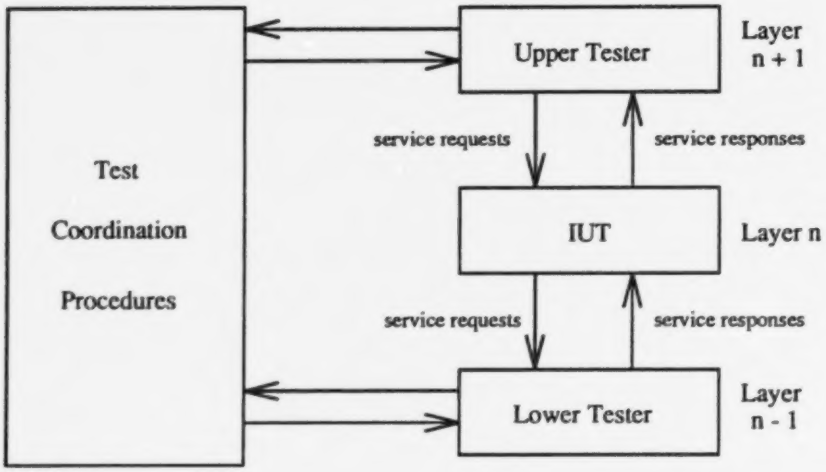

Test
Coordination
Procedures
Upper Tester
Layer n + 1
service requests
service responses
IUT
Layer n
service requests
service responses
Lower Tester
Layer n - 1

Figure 2.1: The Local Method

allow a test system to feed the IUT with inputs and scrutinize the IUT's outputs. The layer which is under test is called layer n and the layer above (below) the IUT as layer $n+1$ $(n-1)$. In order to request services from the IUT, layer $n+1$ $(n-1)$ exchanges service primitives with the IUT at the top (bottom) interface of the IUT. Figure 2.1 shows the architecture of the local test method. Both the upper and lower tester reside in the same system. The upper (lower) tester, which is connected to the upper (lower) interface, controls and monitors events approximate to what the IUT perceives at its upper (lower) interface. Testing is performed by issuing service requests to layer $n-1$ (n) at the lower (upper) interface and validating the service responses at the upper (lower) interface. The Test Coordination Procedures (TCPs) connected to both the upper and lower testers furnish the rules for cooperating between the testers during testing.

2.3.2 Distributed Test Method

The distinction between the local test method and the distributed test method is that the lower interface of the IUT is not exposed. In fact, the IUT and the lower tester are located in two separate units. They are however interconnected through a communication medium. The lower tester is the peer entity of the IUT. The architecture of the distributed system is depicted in figure 2.2. Here, layer *n* conformance is verified by issuing service requests to layer *n* at the upper interface of the IUT and asserting of the service responses at the remote lower tester. The TCPs interact with the two testers and essential information from them is collected. Since the lower tester and the IUT are situated in two different systems, events perceived by them occur at different times. This gives rise to the issues of synchronization and control at the PCOs which must be resolved in order to achieve the test purpose. Linn suggests several assumptions on implementing a distributed test method [17]:

- abstract test cases written for the local method are not applicable; they must be rewritten to reflect the service primitives available to the lower interface of the lower tester;
- the lower tester and IUT are physically separate with the implication that they observe the same test event at different times;
- data loss, delivery out of sequence, and data corruption are possible;
- synchronization and control are more difficult because elements of the test system are distributed over two systems.

Conceptually, the coordinated test method is the same as the distributed test method except that the upper interface of the IUT is not exposed and there exists a Test Management Protocol Data Unit (TMPDU) to maintain and coordinate the

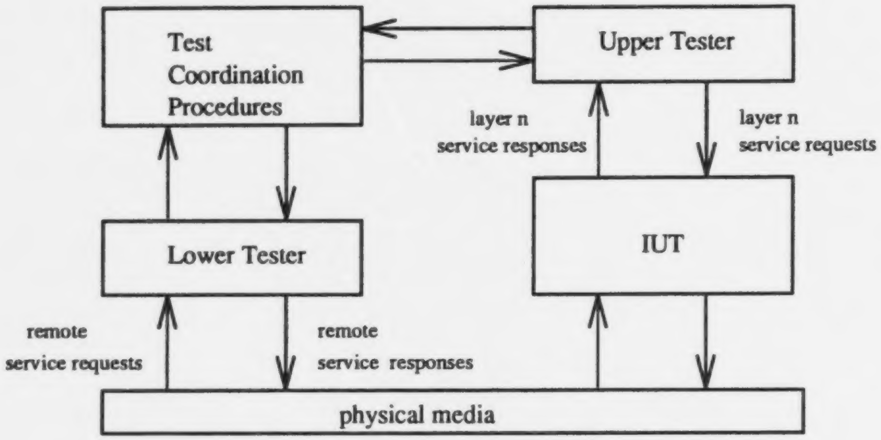

Test
Coordination
Procedures
Upper Tester
layer n
service responses
layer n
service requests
Lower Tester
IUT
remote
service requests
remote
service responses
physical media

Figure 2.2: The Distributed Method

test events. The remote test method has no upper interface at the top of the IUT and is often used for conformance testing of the X.25 protocol.

In general, the error detection potential of a local observer is less powerful than a global observer. However, the complexity of employing a global observer in a multi-entity environment restricts its proficiency. For example, an N-entity network entails N upper testers and one lower tester. To monitor this network, the TCPs must have the ability to synchronize and control the upper and lower testers. In addition, the information accumulated by the TCPs may become enormous. Neither the coordinated test method nor the remote test method were considered in this thesis.

2.3.3 Observers in Real and Simulated Environments

Conformance testing can be conducted in a real or a simulated environment using either passive or active hardware (software) observers as tools.

Hardware observers can test real implementations. An example of testing an implementation using hardware observers is a case study done by Molva *et al.* [2]. The aim of the experiment is to study whether passive hardware observers can detect hardware failures in a fault tolerant medium access control protocol; an impossible task for software observers. Hardware failures that do not have immediate effect on the external bahaviour of the system but whose accumulation can induce errors have been observed [2].

An experiment which used passive software observers to verify a CSMA/CD with Tree Collision Resolution (TCR) protocol implementation was conducted by Berard [3]. The CSMA/CD protocol is programmed using the `lansf` system [7]. This protocol can either be in uncontrolled or controlled mode to express the concept of unslotted and slotted TCR. When in uncontrolled mode, every functioning station has access to the bus. If a collision occurs during the uncontrolled mode, all stations switched to controlled mode. Every station involved in the collision is recorded in a privilege subtree. The claimed properties of controlled mode are as follows:

1. Stations transmit only when they are within the privilege subtree.
2. All stations not participating in an uncontrolled (controlled) collision successfully transmit during the subsequent tournament (sub-tournament).

These observers discovered that property two was violated in the initial implementation. The protocol was then re-implemented using the observer model as the reference. No further errors were discovered by the observers.

Although software observers are unable to validate an implementation in the real environment, they have several advantages: instantaneous knowledge of simulation events, dynamic access to simulation data structures. implementations can be modified easily and cost effectively. With these benefits, software observers become a convenient tool for protocol designing and testing.

Chapter 3

The Distributed Queue Dual Bus Protocol

3.1 Introduction

With increasing demand on sharing data and other resources between Local Area Networks (LANs), another category of communication networks is needed to interconnect them. These new networks which are called Metropolitan Area Networks (MANs), are standardized by the IEEE working group under Project 802.6. The objectives of MANs are to cover areas up to 100 kilometers in diameter, to provide very high transmission rate, in the range of 50 to 150 Mbps, and to support integrated services: data, voice, and video. It is required that the access delay of the MAC (medium access control) protocol employed in MANs should be independent of the size of the network [19].

The Distributed Queue Dual Bus (DQDB) protocol has recently been adopted by

IEEE as a standard to MANs. The protocol, which is designed for the high speed MAN environment to utilize the communication channels efficiently, is based on the Queued Packet and Synchronous Circuit Exchange (QPSX) network proposed by Newman *et al.* [20, 21, 22]. Access to the bus is controlled by a MAC (medium access control) protocol referred to as Distributed Queueing [20], which guarantees bounded access delay for all stations. Under light and medium load, every station has the same share of network bandwidth. Under heavy load, almost no capacity is wasted. All further description of the DQDB protocol presented later in the chapter is based on the proposed standard draft [14]. Note that this draft was further refined before it became a standard. The topology of the DQDB network is delineated in section 3.2. Section 3.3 presents the slot format of the DQDB protocol. The distributed queueing algorithm with one and three priorities is described in section 3.4. Finally, the MAC services provided by the DQDB protocol is detailed.

3.2 The Dual Bus Topology

The *dual* bus topology was originated from the *Fasnet* protocol proposed by Limb *et al.* [15]. The topology is adopted by the DQDB network because of its robustness and high reliability. The dual bus topology consists of two unidirectional buses, a head station at one end of each bus, and stations along the buses. The head stations are responsible for generating *slots* (see section 3.3) to be used by all stations. A DQDB network with 4 stations labeled $0 \cdots 3$ is shown in figure 3.1. Conceptually, each station has an Access Unit (AU) which performs the DQDB layer functions. Each AU has a read port and a write port to both buses. The read port is placed ahead of the write port so that data to be read would not be corrupted by the station's own

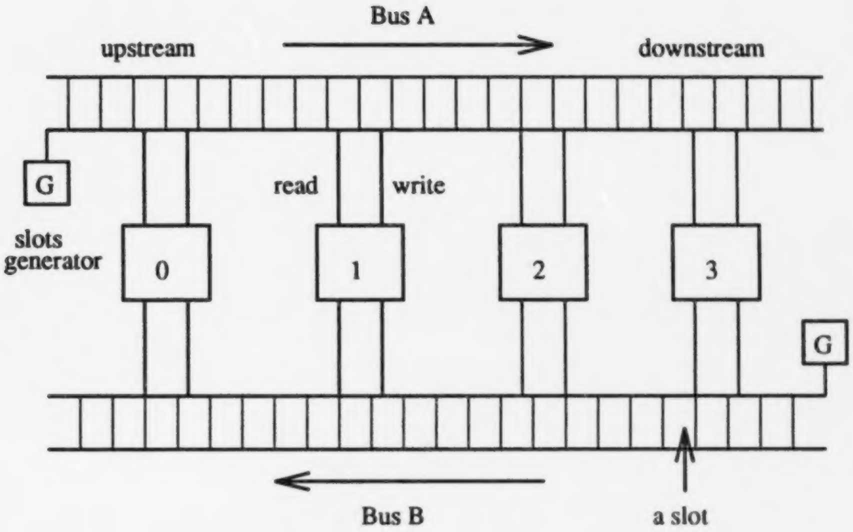

Bus A
upstream
downstream
G
slots
generator
read
write
0
1
2
3
G
Bus B
a slot

Figure 3.1: The Dual Bus Topology

transmission. The buses, denoted bus A and bus B, serve as communication channels for any pair of stations on the network. The operation of the two buses in the network is independent but symmetric. Referring to figure 3.1, if station 0 wishes to transmit messages to station 2, it uses bus A. If station 2 wishes to transmit messages to station 0, it uses bus B. This implies that every station requires to know the relative position of all other stations on the network. The events that trigger a reconfiguration of the network are:

- relocation of a station,
- failure of a bus, or
- failure of a head station.

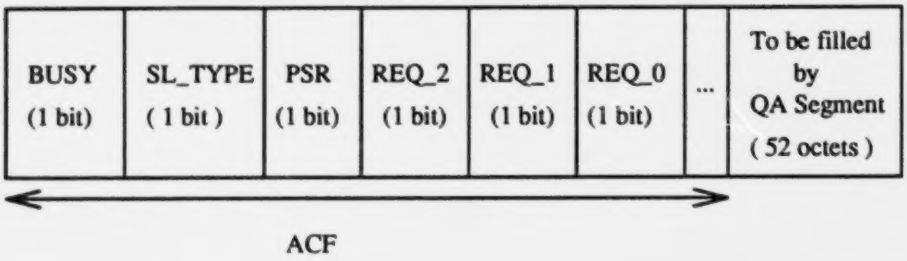


Figure 3.2: The Slot format of DQDB MAN

3.3 Slot Formation

The DQDB network is a slotted system. It provides two types of network accesses: Queued Arbitrated (QA) and Pre-Arbitrated (PA). Pre-arbitrated access is used to support isochronous service (such as, voice transmission), while queued arbitrated access controls non-isochronous service. Both services use fixed length units called *slots* for transmission. In this thesis, we are concerned only with the queued arbitrated access method, as the precise operation of the other traffic is not yet specified in the draft.

The head station of each bus generates empty slots to be used by all stations for sending data and reserving slots. Figure 3.2 shows the slot format of DQDB. A slot consists of two portions: an Access Control Field (ACF) and a data field. The ACF, which is read by every station on the bus, occupies the first octet of every slot. It is where a station can request an empty slot (on the opposite bus). The data portion is where a station can place data to be sent to another station. The total length of a slot is 53 octets. The function of each ACF field is as follows:

1. The BUSY bit shows the status (Empty=0, Full=1) of the current slot. This bit is set once data is placed in this slot and hence not accessible by other stations.

2. The SL_TYPE bit denotes whether the slot is a QA slot (SL_TYPE = 0) or a PA slot (SL_TYPE = 1, which is reserved for future use).
3. The Previous Slot Received (PSR) bit indicates whether the data in the previous slot has been received. If the data has reached the destination, the PSR bit of the subsequent slot is set by the receiver. The concept is to have one or more special stations called *eraser* nodes in the network. An *eraser* node is permitted to buffer an occupied slot until it can see the PSR bit of the subsequent slot. If this bit is set, the node resets the BUSY bit of the stored slot and allows it to be reused. For more information about reuse slots, the reader is referred to [6, 27, 41].
4. The Request field contains three bits (REQ_I where I = 0, 1, 2) which correspond to three levels of priority. The head station of each bus initializes these request bits to zero. If a station has data to transmit (e.g. on bus A), it sets a free request bit of the appropriate priority on bus B. Ultimately, stations situated near the head station (relative to the request station on bus A) will see this request and acknowledge that an additional slot is requested on bus A.
5. The remaining two bits are reserved for future use.

3.4 Distributed Queueing Algorithm

The distributed medium access control queueing algorithm governs the access to the slots on the DQDB bus. Every station keeps track of the current state of the network. When a station has data to transmit, it uses the state information to access the bus. Referring to figure 3.1, let the term *forward* bus denote bus A and *reverse* bus denote bus B. The terms, *upstream* and *downstream*, refer to the relative position of a station to another station on the bus. For instance, station 0 is an *upstream* station of station 1 on the *forward* bus whereas station 3 is a *downstream* station of station 1. Since the operation of the protocol is symmetric, the description of the protocol is based on the *forward* bus only. Section 3.4.1 presents the distributed queueing algorithm for a single traffic. It details how the request and countdown counters are maintained by the distributed queued state machine in assisting a station accesses the bus. Section 3.4.2 describes the operation of the distributed queueing algorithm with three levels of priority. This algorithm explains how the priority requests generated by both the station itself and other stations are accounted.

3.4.1 A Single Priority Level

Every station has two counters: a request (REQ) counter and a CountDown (CD) counter for the *forward* bus (similarly for the *reverse* bus). The REQ counter of a station keeps a record of the number of slots requested by its *downstream* stations. The value of a CD counter is relevant when a station has data for transmission. It indicates the number of empty slot a station has to bypass before it is entitled to access one. The operation of the REQ and the CD counters is managed by the Distributed Queue State Machine (DQSM) of a station. Each station has one machine for each bus.

The machine has two states: *idle* and *countdown*. In both states, the DQSM (considering the *forward* bus) observes the status (busy or empty) of passing slots on the *forward* bus and passing requests on the *reverse* bus. The DQSM is in the idle state when it has no data to transmit. When it is in the idle state:

- The REQ counter is incremented by one when the station detects a request (REQ_0=1) on the *reverse* bus.
- For every empty slot (BUSY=0) passing on the *forward* bus, the REQ counter is decremented by one if it is greater than zero.

The machine remains in the idle state until it has data to send. Upon this time, it sets the value of the CD counter to the value of the REQ counter. Subsequently, it resets the REQ counter to zero, informs the station to send a request and enters the countdown state. The request is sent by setting the next free REQ bit on the *reverse* bus. The slot, which carries this request, propagates towards the end of the *reverse* bus. Eventually, every *upstream* station (relative to the request station on the *forward* bus) sees this request and realizes that an additional slot is requested on the *forward* bus, and thus increments its REQ counter accordingly.

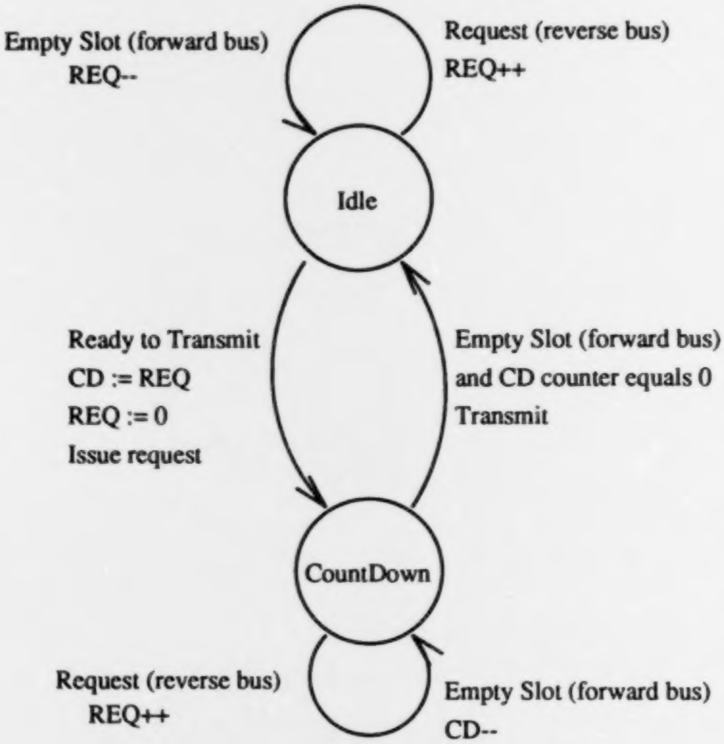
When in countdown state, the machine waits for its turn to access an unused slot on the *forward* bus by performing the following operation:

- The REQ counter is incremented by one for each passing request on the *reverse* bus.
- For every empty slot passing on the *forward* bus, if the CD counter is greater than zero, then the CD counter is decremented by one.
- If the DQSM encounters an empty slot on the *forward* bus, and the CD counter is zero, it sets the BUSY bit of the slot to one and fills the slot with its data.
- After the data is transmitted, the DQSM returns to the idle state.

The data inserted into the slot is called QA segment (see section 3.5.2). The DQSM can only have one outstanding QA segment at a time. The DQSM transition diagram is shown in figure 3.3.

3.4.2 Three levels of Priority

The distributed queueing protocol can be extended to several priorities by having separate distributed queues, REQ counters and CD counters for each priority. With three levels of priority, each station has three REQ_I (I = 0, 1, 2) counters, three CD_I (I = 0, 1, 2) counters and three DQSM_I (I = 0, 1, 2) for each bus. The operation



Empty Slot (forward bus)
REQ--
Request (reverse bus)
REQ++
Idle
Ready to Transmit
CD := REQ
REQ := 0
Issue request
Empty Slot (forward bus)
and CD counter equals 0
Transmit
CountDown
Request (reverse bus)
REQ++
Empty Slot (forward bus)
CD--

Figure 3.3: The DQSM transition diagram

of the counters remains much the same as before, except data of higher priority are taken into account. Similar to the single priority, a DQSM can either be in the *idle* state or the *countdown* state. In both states, all the DQSMs monitor the requests passing on the *reverse* bus and the passing slots (Busy or Empty) on the *forward* bus. A DQSM_I controls the operation of the REQ_I counter and the CD_I counter. Each DQSM starts and remains in the idle state until it has data to send. While in the idle state, the REQ_I counters operate as follows [14]:

- Whenever a REQ_J passes by on *reverse* bus, the REQ_I counter is incremented by one if $I \leq J$.
- If a station generates a request for priority level J, the REQ_I counter is incremented by one if $I \leq J$.
- If an empty slot passes by on the *reverse* bus and a REQ_I counter is not equal to zero, then the REQ_I counter is decremented by one.
- When a DQSM_I has data to be transmitted on the *forward* bus, it shall:
  1. copy REQ_I counter to CD_I counter.
  2. reset REQ_I counter to zero.
  3. issue a request of priority I to be sent on the *reverse* bus.
  4. enter the countdown state.

The DQSM_I remains in the countdown state until the data is transmitted. While in the countdown state, the CD_I counters and the REQ_I counters are maintained by the following actions [14]:

- Whenever a REQ_J passes by on *reverse* bus, the REQ_I counter is incremented by one if $I \leq J$.
- Whenever a REQ_J passes by on *reverse* bus, the CD_I counter is incremented by one if $J > I$.
- If a station generates a request with priority level J, the CD_I counter is incremented by one if $I < J$.
- If an empty slot passes by on the *forward* bus and the CD_I counter is not equal to zero, then the CD_I counter is decremented by one.
- If an empty slot is received on *forward* bus and CD_I counter equals zero, then the DQSM_I sets the BUSY bit of the current slot to one and occupies it with data. Upon completing the transmission, it returns to the idle state.

Each DQSM can only have one outstanding data at a time. This priority access control mechanism allows data of a higher priority to access the bus ahead of data of a lower priority.

3.5 The MAC services to the LLC

The DQDB protocol provides support for transferring a Logical Link Control (LLC) Protocol Data Unit (PDU) from one station to one or more peer stations. The services are handled by the MAC Convergence Function (MCF) block and the Queued Arbitrated Function (QAF) block. A convergence function block is a function or procedure that provides sufficient additional services to enable a layer to support the services expected by a particular higher layer [14]. The MCF block is responsible for the creation of IMPDU (Initial MAC Protocol Data Unit), the segmentation of the IMPDU into fixed length units called DMPDU (Derived MAC Protocol Data Unit), and the reassembly of the IMPDU at the destination station. The QAF block creates a QA segment (which fills the data portion) from the DMPDU to be transmitted by the DQSM. The detailed description on both the MCF block and the QAF block are required in order to reassemble the IMPDU from the QA segments received at the destination. The interaction between the MCF block and the QAF block is shown in figure 3.4. Section 3.5.1 describes the creation of IMPDU. The formation of a QA segment from a DMPDU is presented in section 3.5.2. The last subsection details the reassembly process.

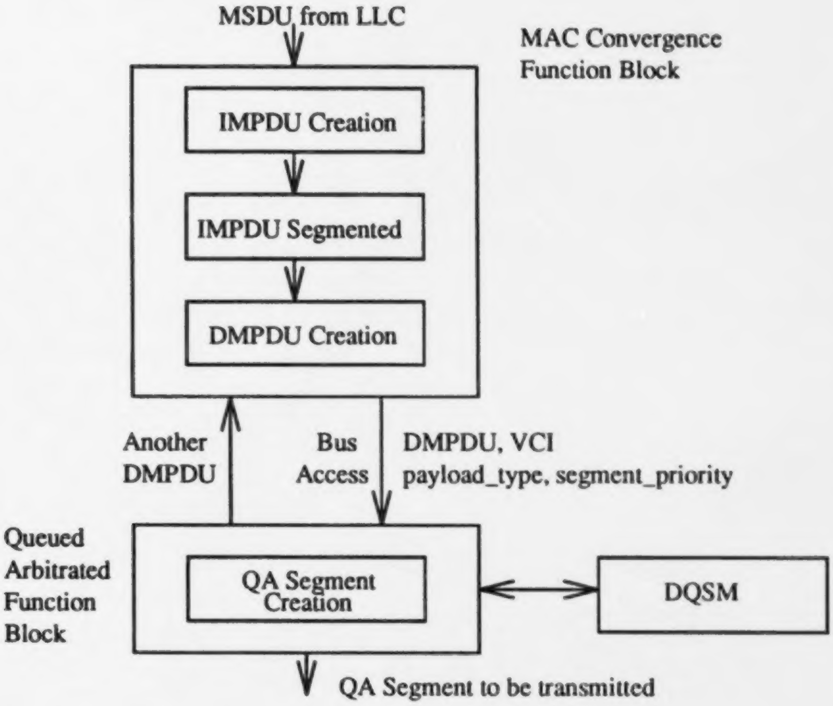

MSDU from LLC
MAC Convergence Function Block
IMPDU Creation
IMPDU Segmented
DMPDU Creation
Another DMPDU
Bus Access
DMPDU, VCI payload_type, segment_priority
Queued Arbitrated Function Block
QA Segment Creation
DQSM
QA Segment to be transmitted

Figure 3.4: The interaction between MCF Block and QAF Block

3.5.1 Creation of IMPDU

Upon receipt of a MAC Service Data Unit (MSDU) from the LLC layer, the MCF block of a source station encapsulates a Common Protocol Data Unit (CPDU) header, a MAC Convergence Protocol (MCP) header, a variable length PAD field, and a Common Protocol Data Unit trailer to the MSDU to form an Initial MAC Protocol Data Unit (IMPDU) as depicted in figure 3.5. The PAD field ensures that the total length of the MSDU field is a multiple of four.

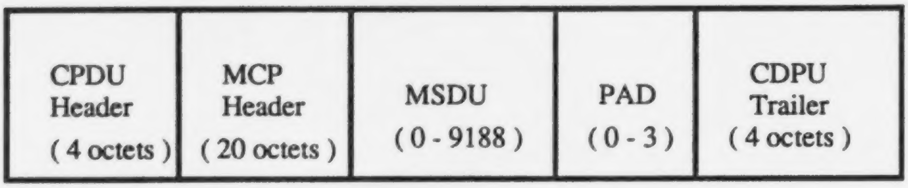


Figure 3.5: The Initial MAC Protocol Data Unit

The CPDU (common protocol data unit) header and its trailer are used to detect the loss of QA segments during the reassembly process at the destination. Figure 3.6 shows the general structure of the CPDU header and its trailer. Both the header and the trailer consist of three subfields. The Beginning-End tag (BEtag) subfield, which is set to the value of the BEtag counter (owned by the MCF block at a source station). The BEtag counter is incremented by one (mod 256) for every new MSDU. The value in the BEtag subfield prevents QA segments derived from different IMPDUs to be concatenated together and should be the same for both the header and the trailer. The Buffer Allocation (BAsize) subfield of the header is set to the length of the MCP header, the MSDU field, and the PAD field. The same value should be inserted into the Length subfield of the CPDU trailer.

The MCP header contains the routing information and the type of service required for the IMPDU. It has five major fields: DA, SA, PI/PL, QOS, and Bridging. The DA field contains the receiver address of the IMPDU. The sender address of the IMPDU is written into the SA field. The PI/PL field is expressed with 2 subfields: a protocol identifier and a PAD length field. The protocol identifier subfield is set to one for LLC service. The value of the PAD length subfield is the number of octets the PAD

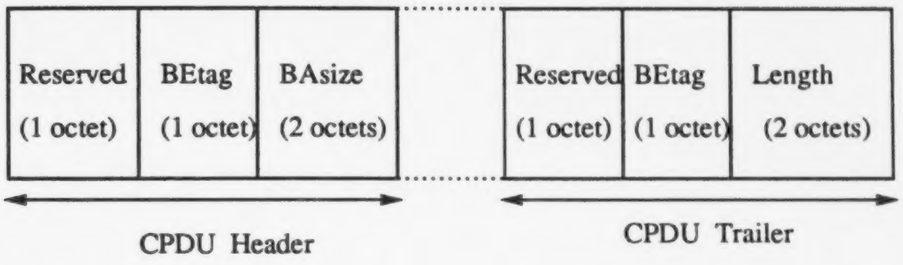

Reserved
(1 octet)
BEtag
(1 octet)
BAsize
(2 octets)
Reserved
(1 octet)
BEtag
(1 octet)
Length
(2 octets)
CPDU Header
CPDU Trailer

Figure 3.6: The structure of CPDU Header and Trailer

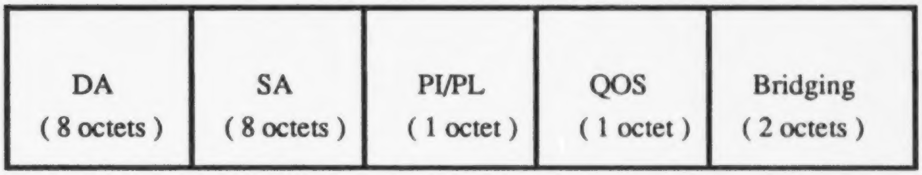

DA
(8 octets)
SA
(8 octets)
PI/PL
(1 octet)
QOS
(1 octet)
Bridging
(2 octets)

Figure 3.7: The structure of MAC Convergence Protocol Header

field occupied in the entire IMPDU. It can be 0, 1, 2, or 3. The remaining two fields are reserved for future use. Figure 3.7 shows the MCP header format.

3.5.2 Creation of DMPDU

An IMPDU is further divided into one or more segmentation units. The length of each segmentation unit is 44 octets or less. Each of these segmentation units is concatenated with a header and a trailer to form a Derived MAC Protocol Data Unit (DMPDU), as shown in figure 3.8. The DMPDU header consists of three subfields. The first is the Segment Type subfield. Segment type of a segmentation unit can be:

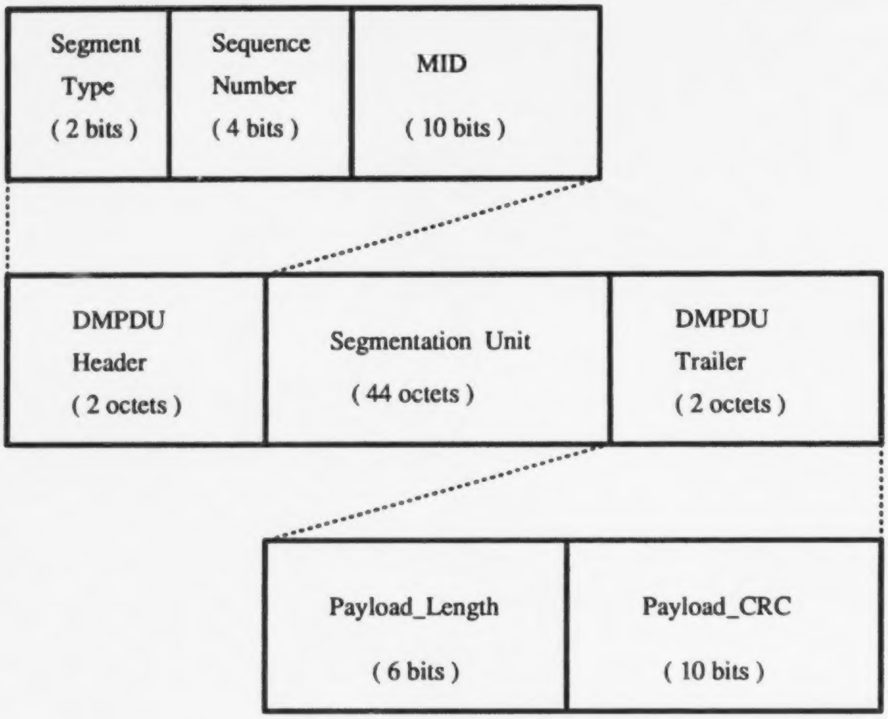

Segment Type (2 bits)
Sequence Number (4 bits)
MID (10 bits)
DMPDU Header (2 octets)
Segmentation Unit (44 octets)
DMPDU Trailer (2 octets)
Payload_Length (6 bits)
Payload_CRC (10 bits)

Figure 3.8: The Derived MAC Protocol Date Unit

Beginning of Message (BOM), Continuation Of Message (COM), End Of Message (EOM), and Single Segment Message (SSM). If an IMPDU forms multiple DMPDUs, the segment type of the first DMPDU is BOM, which signifies the start of a new IMPDU transmission. All subsequent segmentation units derived from the same IMPDU except the last segmentation unit are identified by the COM code in the segment type subfield. The last DMPDU is identified by the EOM code, which indicates the completion of the IMPDU transmission. The SSM as the Segment Type evinces that the total length of the IMPDU is less than or equal to 44 octets.

The Sequence Number subfield and the Message Identifier (MID) subfield are used to identify all the DMPDUs derived from the same IMPDU during the reassembly process. The sequence number of the first DMPDU is assigned to the value of Transmit Sequence Number (TSN) counter (owned by the MCF block). The TSN counter is then incremented by one (mod 16) for each subsequent DMPDU (COM or EOM). The MID is a ten bit subfield. Each station in the network have one or more unique MID(s). The MID is used to provide a logical linking between segmentation units derived from the same IMPDU. Therefore, all the segmentation units derived from the same IMPDU should have the same MID, which is not currently used for sending other IMPDUs. The MID number can be reused by the sender for the next IMPDU transmission immediately after the EOM segmentation unit is sent. The MID subfield of the single segment DMPDU is not assigned to any value.

The DMPDU trailer is expressed with 2 subfields: a Payload length and a Payload CRC. The Payload length subfield indicates the length of the segmentation unit occupying the DMPDU. The value of the Payload length for the BOM and COM DMPDUs is always 44 octets. The Payload length of the last DMPDU is any multiple of 4 in between 4 and 44 octets. For single segment DMPDU, the value of the Payload length is any multiple of 4 in between 28 and 44 octets inclusively. The Payload CRC field provides error detection and error correction in the DMPDU. Figure 3.8 shows the format of DMPDU.

The DMPDU is then passed to the Queued Arbitrated Function (QAF) block. For each DMPDU, the MCF block informs the QAF block which bus the DMPDU should be sent on and the priority level of the DMPDU. The DMPDU is encapsulated with a QA segment header to form a QA segment as shown in figure 3.9. A QA segment

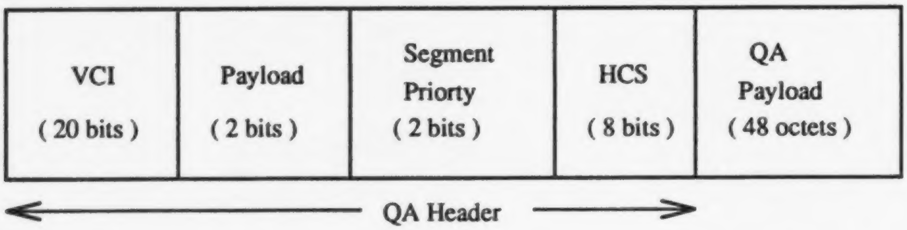


Figure 3.9: The QA Segment

header has four subfields. The first is Virtual Channel Identifier (VCI) is twenty bits long and identifies which function block is responsible for the current QA segment at the destination station (see section 3.5.3). The remaining fields are reserved for future use. The QA segment is then passed to a DQSM (distributed queued state machine) and is ready for transmission.

3.5.3 Reassembly process

The reassembly process takes place at the destination. Every station monitors the passing slots on the bus. Whenever a station encounters an occupied slot (BUSY=1), its QAF block checks the value stored in the VCI subfield of the QA segment. If this value (all bits set to one for QA segment) is recognized by the QAF block, it examines the segment type subfield in the DMPDU header. The subsequent operation of the QAF block depends on the value in the segment type subfield and is as follows:

SSM or BOM It inspects the receiver address in the MCP header. If it matches the address of this station, the QA segment is copied. It records down the Sequence Number and saves the MID onto a list (only if the value is BOM) to receive the remaining multi-segment IMPDUs.

COM or EOM If the MID is currently on the list and the Sequence Number is the expected value, the QA segment is copied. The MID is removed from the list if the value is EOM.

The QAF block then waits for another occupied slot. Since segments are delivered in order (unless corrupted), all the segmentation units are concatenated together to form the original IMPDU if it is a multi-segment IMPDUs. The IMPDU is then verified by the control information contained in the CPDU header and the CPDU trailer. The verification process involves two steps. The first is to compare the value in the Length subfield of the CPDU trailer against the total length of the received IMPDU, a mismatch causes the IMPDU to be discarded. This check is to ensure that all the DMPDUs derived from the same IMPDU are received. In the second step, the value of the BEtag field in the CPDU header is compared against the value of the BEtag subfield in the CPDU trailer, a mismatch causes the IMPDU to be discarded.

Chapter 4

SMURPH — An Overview

4.1 Introduction

`Smurph` has been developed at the University of Alberta by P. Gburzyński and P. Rudnicki. The predecessor of `smurph`, `lansf` (Local Area Network Simulation Facility) [7] is programmed in C and has been successfully applied to investigate the performance and correctness of several protocols [8, 9, 10, 12]. `Smurph`, evolved from `lansf`, has been programmed in C++ and runs under the Unix[1] operating system. It can be viewed as an implementation of a certain executable protocol specification language [11]. A protocol specified in `smurph` can be executed by the `smurph` simulator. The underlying details of the simulator are hidden from the user. All the simulation-related operations, such as creating and scheduling individual events, are maintained under a high-level interface. Furthermore, the `smurph` modeling environment is able to reflect all the relevant phenomena occurring in real physical channels.

---

[1] Unix is a registered trademark of AT&T Bell laboratories

An extensive description of **smurph** can be found in [11]. To start with, a sample code of the **smurph** specification language is presented. Then, the fundamental concepts of **smurph** are described.

4.2 A Simple Modeling Example

This section presents a part of the DQDB implementation developed in this thesis. The DQDB specification can be expressed in terms of several interacting processes; the code of one of these processes named `Monitor` (each station owns a copy of this process) is shown below:

```
process Monitor (Node) {

     // Local Variables Declaration
     Port      *MyPort;
     QASegment **PktPtr;
     ACF       **SlotPtr;

     void setup () {
        MyPort      = S->PortLR;
        PktPtr      = &(S->PktPtrLR);
        SlotPtr     = &(S->SlotHdrLR);
        }
     states {WaitSlot, SlotStatus, SlotHeader, SlotBody};

     perform {
        state WaitSlot :
               MyPort->wait(BOT, SlotCheck);

        state  SlotStatus :
               if (ThePacket->TP == ACF) {
                        *SlotPtr = TheSlot;
                        proceed (SlotHeader);
                }
                else
                        proceed (SlotBody);

        state SlotHeader :
                if (!(*SlotPtr)->REQ_0)
                        S->REQ_NOT_SET->put ();
                else
                        S->REQ_SET->put ();
                if (!(*SlotPtr)->BUSY)
                        S->EMPTY_SLOT->put ();
                skipto (WaitSlot);

        state SlotBody :
                *PktPtr = TheSegment;
                S->RX_BUS->put ();
                skipto (WaitSlot);
        }
}
```

The purpose of the `Monitor` process is to detect the arrival of a slot at one of the `read` ports owned by the station and notify the appropriate processes (of this station) according to the status of the slot. The local attributes, `MyPort`, `PktPtr` and `SlotPtr`, are set by the `setup` function to the port connected to the *forward* bus, and to the variables of the station (`PktPtrLR` and `SlotHdrLR`) for storing the address of the QA (Queued Arbitrated) segment and the address of the ACF (Access Control Field) respectively. The `S` attribute is pointing at the station which owns this process. Thus, `S->REQ_NOT_SET`, `S->REQ_SET`, `S->EMPTY_SLOT` and `S->RX_BUS` identifies the mailboxes declared at the station for communication between this process and other station processes. The possible states which the `Monitor` can be in at any particular time is listed in `states`. Each state defines a sequence of operations in the `perform` method. Transition from one state to another are done by invoking one of the 3 operations: `wait`, `proceed`, `skipto`.

When the `Monitor` is created, its local attributes are initialized first before entering the state `WaitSlot`. In this state, it issues a wait request to the associated port specifying that it wants to be restarted by the earliest `BOT` (Beginning Of Transmission) event—the beginning of a slot. Note that an empty slot triggers only one `BOT` whereas an occupied slot activates two `BOT`s: one by the ACF and the other by the QA segment (see figure 5.2). Whenever a slot arrives, the process wakes up at state `SlotStatus`. Here, the first operation is to determine what caused the `BOT` event. The variable `ThePacket`, which belongs to the *process environment* (whose purpose is to pass information related to the restarting event to the awakened process), points to the packet which has triggered the `BOT` event. If it is the ACF, it proceeds to state `SlotHeader` and examines the attributes of the ACF to inform the appropriate processes (by putting a signal into the `REQ_NOT_SET` mailbox—if request bit is not set, `REQ_SET` mailbox—if request bit is set, `EMPTY_SLOT` mailbox—if unused slot). The `skipto` operation is called to explicitly request an ITU (time unit of `smurph`) delay before control returns to state `WaitSlot`. This delay is necessary to ensure that the `BOT` event will not be sensed by this process again. Otherwise, the process would operate infinitely on the same `BOT` event without the simulation time ever advancing. If the `BOT` is caused by an arriving QA segment, `Monitor` continues at state `SlotBody`. Before returning to state `WaitSlot`, it saves the address of the QA segment and notifies the `receiver` process by depositing a signal in the `RX_BUS` mailbox. Figure 4.1 shows the transition diagram of the `Monitor` process.

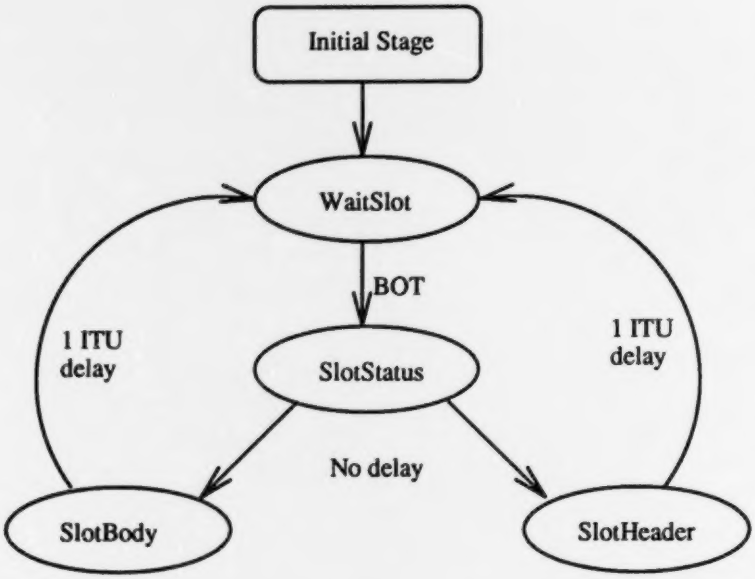

Initial Stage
WaitSlot
BOT
SlotStatus
1 ITU delay
1 ITU delay
No delay
SlotBody
SlotHeader

Figure 4.1: The transition diagram of the `Monitor` process

4.3 The Modeling Environment

This section discusses some of the features provided by **smurph** for modeling distributed systems. In **smurph**, all the network components such as stations, links, and packets are referred to as objects. An object is either an instance of a base type (as class in C++) or a type derived from a base type. Objects can be further categorized as *passive* or *active*. Passive objects are usually handled by active objects. For example, a `client` (active) can remove a packet (passive) from a message queue of a station. The `client` belongs to a special category of active objects called **`Activity-Interpreters`** which accept activities from the station processes (e.g. transmit a packet from a port to a link) and turn them into future events.

4.3.1 Time

Time in **`smurph`** is partitioned into equal size intervals. Each of these intervals is called an *Indivisible Time Unit* (ITU). If multiple events occur in the same ITU, one of them is randomly chosen to be processed. The precision of time is selected by the user. Although there is no explicit limit on this precision, the execution speed is curtailed with a higher resolution of time. During a simulation experiment, statistical results are gathered according to another time unit called *Experimenter Time Unit* (ETU)—to make the results more legible for users to analyze. An ETU can be expressed in terms of ITUs and vice versa.

4.3.2 Network Topology

A network topology is defined in terms of a set of stations and a set of links interconnected via ports. Stations, are objects belonging to type `Station` (defines a station skeleton). This skeleton can be extended by adding additional characteristics to form the actual stations. The services of a station are maintained by a collection of processes belonging to it.

Typically, the interface between a link and a station is provided by a port. Usually, a port is created within the context of a specific station. It permits the station to start and end a packet transmission to the link. Events such as beginning or end of packet transmissions, beginning or end of jamming signals, and collision of packet transmissions can be detected at the port. The *transmission* rate attribute of a port specifies the amount of time (in ITUs) required to send a single bit into the port.

A link models a communication channel such as a fiber optic carrier. It allows stations connected to the same link to exchange packets. **Smurph** provides two basic link concepts: unidirectional and bidirectional link models.

4.3.3 Traffic Patterns

The traffic patterns (objects of type `Traffic`) model the distribution of traffics in a network. There can be several traffic patterns in a single simulation experiment. Each traffic pattern generates new messages according to a specific format and queues these messages at the sending stations. The distribution parameters of a traffic pattern allow the user to specify the distribution of senders or receivers (the probability of being an intended sender or receiver), message inter-arrival time and message length. The message inter-arrival time can either be exponentially distributed or uniformly distributed. Similarly, length can either be exponentially distributed or uniformly distributed. A `client` *AI*, which can be viewed as a union of all traffic patterns, notifies a station upon arrival of a new message.

4.3.4 Processes

A **smurph** specification defines a collection of extended finite state machines in which state transitions are triggered by various events generated by the network environment. A **smurph** process (an object of type `Process`) forms a finite state machine. Thus, the behaviour of a protocol is simulated by a set of processes. Each process can reference the variables of the station owning them. A process type has three main components:

- a collection of local variables,
- a `setup` method, and
- a `perform` method.

When a process is created, local variables are initialized by the `setup` method. The `perform` method contains the code of the process. A newly created process enters the initial state automatically. It can then be either in the *waiting* state or in the *transition* state. A process goes to a *waiting* state by invoking a `wait` function which specifies the future events that the process wants to perceive such as, beginning of transmission, receiving signals from other processes, etc. A process can anticipate more than one future event by executing more than one `wait` request. The earliest occurrence of one of the awaited events wakes up the process. If multiple awaited events occur in the same ITU, one of them is randomly chosen to be processed and the process wakes up in the corresponding state. A process is said to be in a *transition* state when it is executing its code. During the *transition* state, no simulation time elapses; simulation time only flows while the process is in a waiting state—as the run-time system is processing other events. When a process is in a *transition* state, variables can be manipulated, decisions can be made, new processes can be created, signals can be sent and activities can be initiated. The life cycle of a `smurph` process is shown in figure 4.2. A process can create another process and new processes are created with the `create` operation. All station processes are created by an initial root process. Termination of a process is done by invoking the `terminate` function.

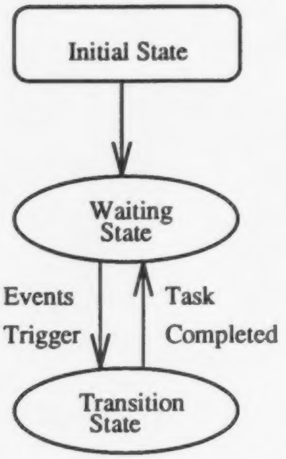


Figure 4.2: The life cycle of a **smurph** process

4.3.5 Process Synchronization

Process communication and synchronization in **smurph** is accomplished by passing signals and exchanging messages (compound objects). The latter facility is furnished by *mailboxes.* Mailboxes provide dynamic and efficient communication mechanism between processes. A mailbox is always associated with a station. A station process can deposit a message into a mailbox, remove a message from a mailbox, peek at the first message of a mailbox and request the status (Full, Empty, Present) of a mailbox. The capacity of a mailbox is specified at its creation. Depending on how it is defined and created (modified by the user), a mailbox may behave as a simple signal passing device (capacity is less than or equal to one) or a FIFO-type storage called queued mailbox (capacity is greater than one). New messages are not accepted once a mailbox becomes full. Another way of communicating between processes is to pass signals among themselves directly without referring to a mailbox.

Chapter 5

The Implementation of DQDB in SMURPH

5.1 Introduction

This chapter describes the implementation aspects of the Distributed Queue Dual Bus (DQDB) protocol presented in Chapter 3. The station types for building DQDB stations are presented in section 5.2. Section 5.3 describes how the network topology is constructed. The traffic condition is defined in section 5.4. Section 5.5 details the operation of each station process. The summary of this chapter is presented in section 5.6. The complete DQDB source code is listed in Appendix A.

5.2 Station Types

The DQDB network consists of two different types of stations: regular stations (assumed homogeneous), and head stations. A head station, located at the head of a bus, behaves like a regular station and also acts as the slot generator for the bus. The following declaration defines the two station types:

```
station Node {
   Port       *PortLR, *PortRL;
   Mailbox    *Mb.DONE, ... declaration of other mailboxes ...
   ACF        *SlotHdrLR, *SlotHdrRL;
   QASegment  Buffer_0_LR, Buffer_1_LR, Buffer_2_LR,
              Buffer_0_RL, Buffer_1_RL, Buffer_2_RL,
              MacBuf_0, MacBuf_1, MacBuf_2,
              *PktPtrLR, *PktPtrRL;

   void setup () {
              PortLR = create Port (TRate);
              PortRL = create Port (TRate);
              Mb.DONE = create Mailbox (1);
              ... create other mailboxes ...
              clearFlag(Buffer_0_LR.Flags, PF_full);
              ... clear other buffers ....
              }
};

station Head: Node {
   ACF        SlHeader;
   void setup () { Node::setup(); }
};
```

A regular station is an object belonging to type `Node`, which in turn is derived from the base type `Station` with added attributes. The `Head` type inherits all the attributes from `Node` type and declares one private attribute—a buffer for storing an

ACF (initialized by the `slotter` process see section 5.5.1). Each station has two ports: one to the *forward* bus (pointed to by `PortLR`), the other to the *reverse* bus (pointed to by `PortRL`). Both ports are for reading data from the bus and transmitting data onto the bus. There are 66 mailboxes created at each station for process synchronization. These mailboxes serve as simple signal passing devices, and thus are only capable of storing one pending signal. The variable `SlotHdrLR` (`SlotHdrRL`) is for saving the address of an ACF arrived at `PortLR` (`PortRL`). Within a station, there are 9 buffers: `MacBuf_I`, `Buffer_I_LR` and `Buffer_I_RL` (I = 0, 1, 2) for storing segments from three priorities. The `MacBuf_I` buffers are used to store a segmentation unit of priority I at the MCF (MAC Convergence Function) block. Eventually, the segmentation unit forms a QA segment and is ready to be forwarded to its destination. If it is to be transmitted onto the *forward* bus, it is deposited into the `Buffer_I_LR` buffer; otherwise it is stored in the `Buffer_I_RL` buffer.

Upon the creation of a station object, its `setup` function creates ports and mailboxes. Note that all the stations have the same transmission rate (`TRate`—read from a file). The function then empties all the buffers.

5.3 Network Configuration

The network configuration is defined by the following function:

```
initTopology () {
  int    NNodes;
  long   SegLen, BusLength;
  PLink  *LinkLR, *LinkRL;
  Node   *Prev, *Curr;

  readIn (NNodes); readIn (BusLength);
  create Head;
  for (int i=1; i<NNodes-1; i++) create Node;
  create Head;

  SegLen = BusLength / (NNodes-1);
  LinkLR = create PLink (NNodes);
  LinkRL = create PLink (NNodes);
  for (Prev=NULL, i=0; i<NNodes; i++, Prev=Curr) {
       TheStation = Curr = (Node*) idToStation (i);
       Curr->PortLR -> connect (LinkLR);
       if (i>0) Prev->PortLR -> setDTo(Curr->PortLR, SegLen);
  }
  for (i=NNodes-1; i>=0; i--, Prev=Curr) {
       Curr = (Node*) idToStation (i);
       Curr->PortRL -> connect (LinkRL);
       if (i<NNodes-1) Prev->PortRL -> setDTo(Curr->PortRL, SegLen);
  }
}
```

The function starts with reading in the number of stations (`NNodes`) and the length of a bus (`BusLength`). We assume that stations are uniformly distributed along the bus, and the distance between any pair of stations is saved in `SegLen`. The first loop creates two head stations and `NNodes`-2 regular stations. Then two unidirectional links (`LinkLR` and `LinkRL`) with `NNodes` ports are created. The `NNodes` ports provide the interface between the stations and the links. The link `LinkLR` denotes the *forward* bus, whereas the link `LinkRL` denotes the *reverse* bus. The next (last) loop connects all the stations to the `LinkLR` (`LinkRL`) and assigns the distances (`SegLen`) between all pairs of stations on the link.

5.4 Traffic definition

The traffic condition in the network is defined by the following function:

```
initTraffics () {
   double mit, MesLen;

   readIn (mit); readIn (MesLen);
   UTPat = create UTraffic (SCL_off+MIT_exp+MLE_unf, mit,
                                    MesLen, MesLen);
   UTPat->addSender ();  UTPat->addReceiver ();
   readIn (mit);
   U1TPat = create UTraffic (SCL_off+MIT_exp+MLE_unf, mit,
                                     MesLen, MesLen);
   U1TPat->addSender ();  U1TPat->addReceiver ();
   readIn (mit);
   U2TPat = create UTraffic (SCL_off+MIT_exp+MLE_unf, mit,
                                     MesLen, MesLen);
   U2TPat->addSender ();  U2TPat->addReceiver ();
}
```

Three traffic patterns (pointed to by the attributes: `UTPat`, `U1TPat` and `U2TPat`) are established to represent three levels of priority. All three traffic patterns are instances of type `UTraffic`, which is derived from the base type `Traffic` with specific format (`QASegment`). The procedure starts by reading in the mean message inter-arrival time (`mit`) and the length of the message (`MesLen`). Each traffic pattern is created with a different mean value. The parameter `SCL_off` permits the users to suspend a traffic pattern from generating messages during the network configuration phase. When the network is fully running, the users inform the suspended traffic pattern to resume its duty. The mean inter-arrival time of all traffic patterns are exponentially distributed and the lengths of the messages are uniformly distributed.

With the above declaration, all stations receive messages with equal probabilities, and contribute the same amount of traffic to the network load.

5.5 The Protocol Code

The services of each DQDB station are basically maintained by the MAC Convergence Function (MCF) Block and the Queued Arbitrated Function (QAF) block (see figure 3.9). Although it is possible to express both function blocks in a single process, the code generated in this way can be difficult to debug. A rational method is to distribute the services among several interacting processes, thus exposing the real semantics of the system. Since the operation of the MCF block is sequential, it forms a single process. The internal structure of the process is sub-divided into a series of states where the formation of each DMPDU (e.g. BOM, COM, EOM) is tested by observers (see section 6.3.5). To express the QAF block in terms of processes, we take into consideration the priority traffics and buses. According to the proposed draft [14], six DQSMs (Distributed Queue State Machines) and six RQSMs (Request Queue State Machines) for three priority traffics and two buses are used. Routines such as monitoring slots, creating QA segments from DMPDUs, transmitting QA segments and receiving QA segments form a `Monitor` process, a QAP (Queued Arbitrated Portion) process, a `Transmitter` process and a QAR (Queued Arbitrated Receive) process respectively. Occasionally, the QAR process creates a child process called Reassembly State Machine (RSM) to reassemble the entire IMPDU from the received QA segments. The RSM process disappears after it has completed its task. With this layout, we are able to check the internal consistency of a station by validating all these processes using observers. A part of our implementation of DQDB including the creation of protocol processes is listed below.

```
process Root {
   UTraffic *UTPat, *U1TPat, *U2TPat;
   ....
   initTopology();  initTraffics();
   for (int i = 0; i < NStations; i++) {
        TheStation = idToStation (i);
        if (i == 0) create Slotter (RIGHT);
        if (i == NStations-1) create Slotter (LEFT);
        create MCF_Block ();
        create QAS_Block (RIGHT); create QAS_Block (LEFT);
        create QAR_Block (RIGHT); create QAR_Block (LEFT);
        create Monitor (RIGHT);   create Monitor (LEFT);
        create DQSM (RIGHT, Priority_0);
        create DQSM (RIGHT, Priority_1);
        create DQSM (RIGHT, Priority_2);
        create RQSM (RIGHT, Priority_0);
        create RQSM (RIGHT, Priority_1);
        create RQSM (RIGHT, Priority_2);
        create Transmitter (RIGHT, Priority_0);
        create Transmitter (RIGHT, Priority_1);
        create Transmitter (RIGHT, Priority_2);
        create DQSM (LEFT, Priority_0);
        create DQSM (LEFT, Priority_1);
        create DQSM (LEFT, Priority_2);
        create RQSM (LEFT, Priority_0);
        create RQSM (LEFT, Priority_1);
        create RQSM (LEFT, Priority_2);
        create Transmitter (LEFT, Priority_0);
        create Transmitter (LEFT, Priority_1);
        create Transmitter (LEFT, Priority_2);
    }
    ....
}
```

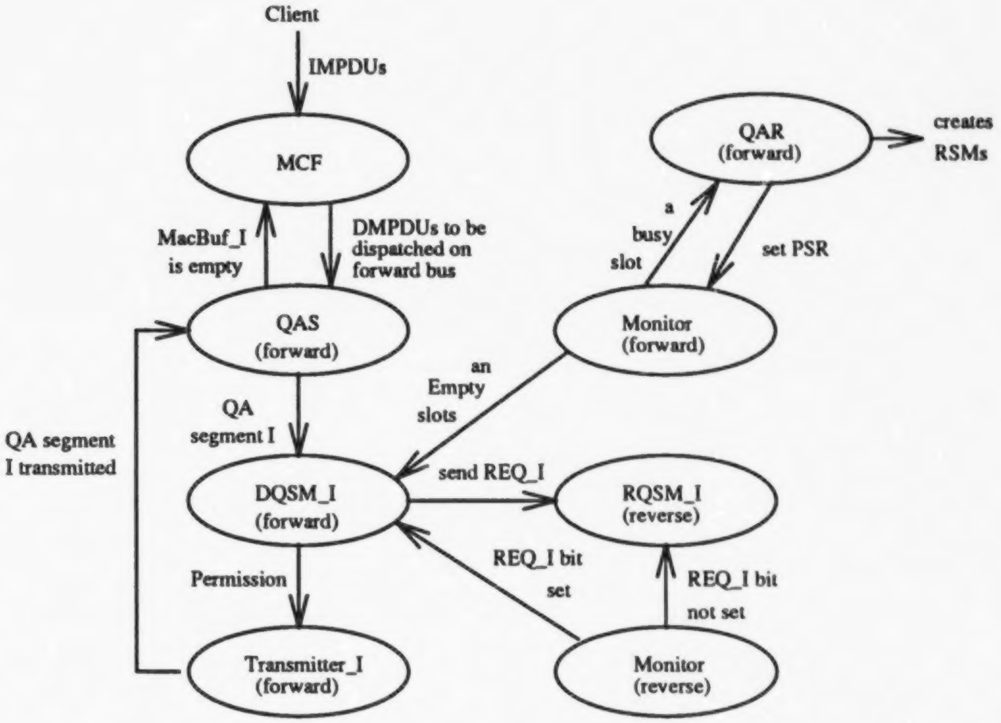

Client
IMPDUs
MCF
MacBuf_I is empty
DMPDUs to be dispatched on forward bus
QAS (forward)
QA segment I
QA segment I transmitted
DQSM_I (forward)
Permission
Transmitter_I (forward)
QAR (forward)
creates RSMs
a busy slot
set PSR
Monitor (forward)
an Empty slots
send REQ_I
RQSM_I (reverse)
REQ_I bit set
REQ_I bit not set
Monitor (reverse)

Figure 5.1: The interaction between station processes

Before the **`create`** operations are invoked, the environmental attribute **`TheStation`** is set to point to the station which will own the processes to be created. The parameters passed to the **`setup`** function of a process identify the bus (`RIGHT` —*forward*) and the priority of the process. Figure 5.1 shows how the processes interact with each other. Note that the **`slotter`** process is missing from the diagram since it is a non-interacting process.

5.5.1 Slotter Processes

A `slotter` process, retained only by a head station, transmits ACFs (Access Control Field) on the selected port of the head station (`PortLR` on *forward* bus and `PortRL` on *reverse* bus) at time duration of 52 octets. Internally in `smurph`, an ACF is an instance of type `Packet` augmented with user defined attributes: BUSY, SL_TYPE, PSR and REQ_I (I = 0, 1, 2). The standard attributes of this type include traffic pattern (`TP`), the sender and receiver addresses, the information length of the packet (`ILength`), the total length of the packet(`Tlength`) and the status of the buffer (`Flags`). Upon creation, the process initializes all the attributes of the `SlHeader` buffer, which stores an ACF, as follows:

1. All the user defined attributes are set to `NO`. A `NO` value means a field has not been set yet.
2. The `TP` is set to `SLOT` to differentiate an ACF from a QA segment whose `TP` can be 0, 1 or 2.
3. Both the `Tlength` and `Ilength` are set to 8 bits since an ACF carries no extra information.
4. The receiver and sender addresses are assigned to `NONE` as they serve no purpose.
5. The `PF_FULL` code indicates the buffer is full and its content (an ACF) is ready to be sent. The buffer is always full unless it is explicitly cleared by the process.

ACF transmission starts in state `StartSlot`. When the transmission is concluded, the `slotter` wakes up at state `SlotDelay`. Having terminated the ACF transmission, it sets up an alarm clock for `417 + x` bits [the length of a QA segment plus (1+x) safety bits]. This period is referred to as a silence period. When the alarm clock goes off, the process returns to state `StartSlot`. The whole operation is repeated. The safety bits serve as an interval between a QA segment and a successive ACF, as

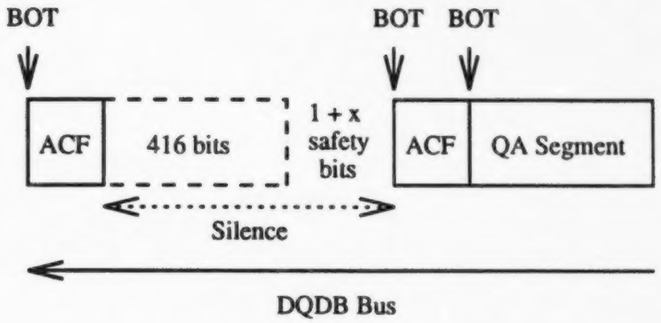


Figure 5.2: The safety bits

shown in Figure 5.2. This is essential to avoid a station's transmission from colliding with the subsequent ACF due to the local clock error. The value of x, which can range from 0 to 1000 ITUs, is supplied by a random function. Another aspect of introducing the safety bits is to determine whether the unsynchronize slot generators has any effect on the protocol since slot generators are never synchronized in real situations.

5.5.2 MAC Convergence Function process

The main task of the MCF process is to divide the newly attained IMPDU (Initial MAC Protocol Data Unit) from the `Client` into one or several DMPDUs (Derived MAC Protocol Data Unit) and transfer them to the Queued Arbitrate Portion (QAP) process. Note that the subfields of the CPDU (Common Protocol Data Unit) header, the MCP (MAC Convergence Protocol) header and the CPDU trailer have not been set (refer to figure 3.5). The local variables: `MacBuf_I`, `Msg_Len_I`, `BEtag_I`, and

`Tx_Seq_Num_I` (I = 0, 1, 2) are to identify the station's buffers, to record the message length of priority I, to serve as beginning-end tag counters and as transmit sequence number counters respectively.

The MCF process consists of 9 states. The operation cycle starts in the state `WaitIMPDUs` where the process suspends itself by invoking the following wait requests:

```
state WaitIMPDUs :
                U2TPat->wait (ARRIVAL, Prior2);
                U1TPat->wait (ARRIVAL, Prior1);
                UTPat->wait  (ARRIVAL, Prior0);
```

If an IMPDU of priority I arrives at the station, the MCF wakes up at state `PriorI`. Having acquired the first 44 octets of the IMPDU the process advances to state `FDMPDU` and starts setting all the subfields in the various headers (CPDU, MCP, DMPDU) to appropriate values. The length of the MSDU (MAC Service Data Unit—IMPDU information portion) is saved in the local attribute `Msg_Len_I`. If it is a single segment IMPDU, the MID (Message IDentifier) subfield is set to zero and control proceeds to state `LDMPDU`. Otherwise, the MID subfield is set to the sum of the station's serial number and the number of stations multiplied by the priority of the IMPDU. In this case, every station has a unique MID for each traffic. The process continues at state `WaitAccess`.

Knowing the priority of the DMPDU and the bus on which it is to be transmitted, the MCF sends a signal to the proper QAP (by putting a dummy item into the appropriate mailbox), and thus informs the QAP that the DMPDU is ready to be fetched. Consequently, the process branches to state `WaitStates` and issues the following wait requests:

- If all the `MacBuf_I` (I = 0, 1, 2) are full, a successful fetch of a DMPDU_I by the QAP wakes up the process. Control shifts to state `DoneI`.

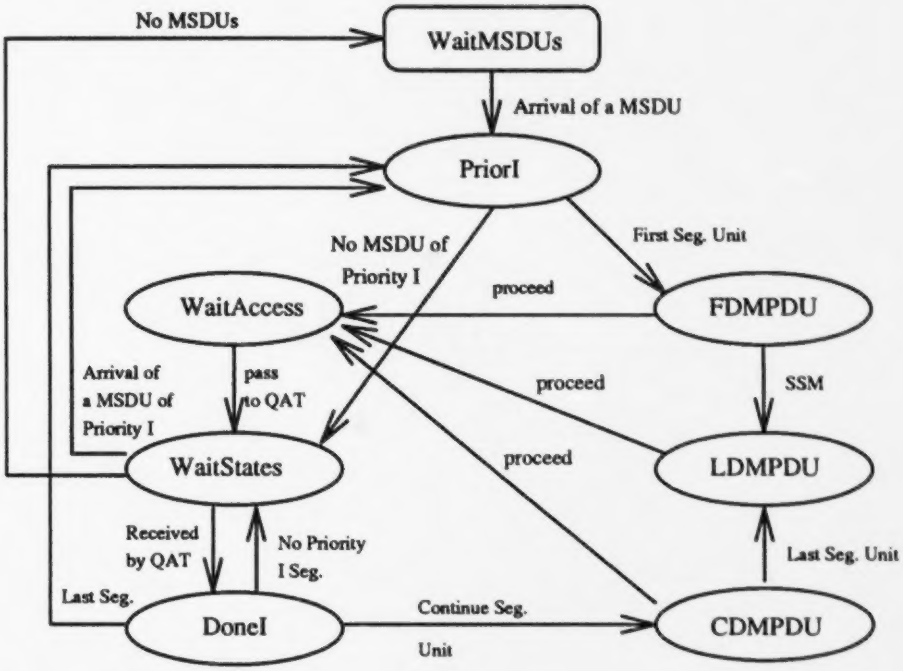

No MSDUs
WaitMSDUs
Arrival of a MSDU
PriorI
No MSDU of Priority I
First Seg. Unit
WaitAccess
proceed
FDMPDU
Arrival of a MSDU of Priority I
pass to QAT
proceed
SSM
WaitStates
proceed
LDMPDU
Received by QAT
No Priority I Seg.
Last Seg. Unit
Last Seg.
DoneI
Continue Seg. Unit
CDMPDU

Figure 5.3: The MCF transition diagram

- If all the MacBuf_I are not filled, either a successful fetch of a DMPDU_I by the QAP or the arrival of an IMPDU to be stored in `MacBuf_J` (where J is empty) wakes up the process. Control continues at state `PriorI`.

The remaining DMPDUs of the IMPDU are acquired in state `DoneI`. Having incremented the counters: `BEtag_I` and `Tx_Seq_Num_I`, the MCF is in state `CDMPDU` if the attempt to obtain the next DMPDU is successful; otherwise it is in state `WaitAccess`. In the state `CDMPDU`, the header of a continuation DMPDU are initialized. Then, the MCF can be in either state `WaitAccess` or state `LDMPDU`. The latter takes place when the DMPDU is the last segment of the IMPDU. Figure 5.3 shows the states transition of the MCF process.

5.5.3 Queued Arbitrated Portion process

The QAP process is concerned with creating a QA segment from the DMPDU received from the MCF process. Every station has two QAP processes; one on each bus (considering *forward* bus here). The process starts at state `WaitSignals` and waits for the `ACC_I` signals from the MCF. If a DMPDU of priority I is received, the process wakes up at state `AccessI` (I = 0, 1, 2). In this state, the subsequent action of the QAP depends on the status of the `Buffer_I_LR` buffer:

Empty The QAP stores the DMPDU, signals the MCF that it has successfully fetched the DMPDU and continues to state `MovetoXtmBuf`. Here, it encapsulates a QA Header to the DMPDU and signals the DQSM_I about the newly created QA segment before advancing to state `WaitState`.

Full Control goes to state `WaitStates` where the process suspends itself and waits for the `Buffer_I_LR` to be emptied by the `Transmitter` process or the arrival of a DMPDU to be stored in the `Buffer_J_LR` (where J is empty). Control transfers to the state `AccessI` once the process wakes up.

A transition diagram of the QAP process is depicted in figure 5.4.

5.5.4 Monitor process

Each station owns two `Monitor` processes: one for monitoring the *forward* bus (through `PortLR`), and the other for monitoring the *reverse* bus (through `PortRL`). Let us consider the `Monitor` process of the *forward* bus. The process has nothing to do until it is awakened by the `BOT` event at its port. In such case, the `Monitor` moves to state `SlotStatus` to determines what activated the `BOT` event. If it is a QA

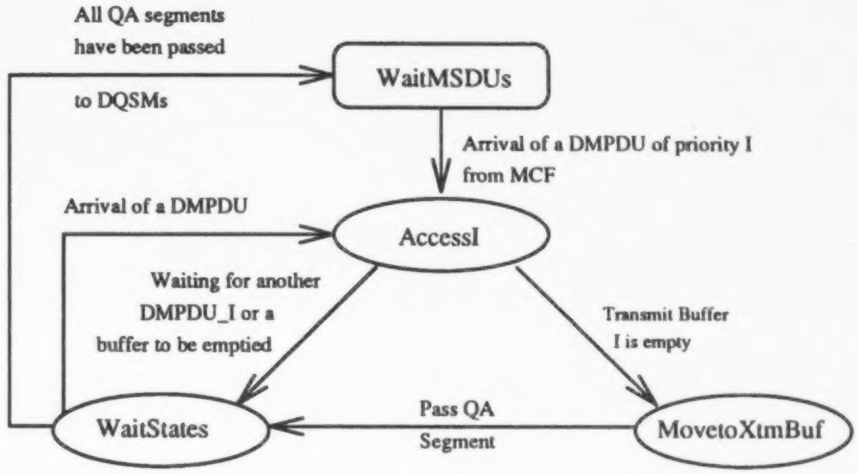

All QA segments
have been passed
to DQSMs
WaitMSDUs
Arrival of a DMPDU of priority I
from MCF
Arrival of a DMPDU
AccessI
Waiting for another
DMPDU_I or a
buffer to be emptied
Transmit Buffer
I is empty
Pass QA
Segment
WaitStates
MovetoXtmBuf

Figure 5.4: The QAS transition diagram

segment, it puts a dummy item in the `RX_BUS_LR` mailbox of the Queued Arbitrated Receive (QAR) process. When the `BOT` is triggered by an ACF, the `Monitor`:

- Sets the PSR = `YES`, if the previous QA segment is received.
- Signals the DQSM_J (Distributed Queue State Machine) on the *reverse* bus by depositing a dummy item in the `REQ_SET_J_RL` mailbox, if the REQ_I ($I \geq J$) bit is set.
- Signals the RQSM_I (Request Queue State Machine) on the *forward* bus by depositing a dummy item in the `REQ_NOT_SET_I_LR` mailbox, if the REQ_I bit is not set.
- Signals all the DQSMs on the *forward* bus by putting a dummy item in each of the `EMPTY_SLOT_I_LR` mailbox, if the BUSY bit is not set

5.5.5 Distributed Queue State Machine

This section explains the implementation details of the DQSM presented in Section 3.4. There are six DQSMs, one for each combination of buses and priorities, within a station. Each DQSM consists of two local attributes: `REQ_cntr` (request counter) and `CD_cntr` (countdown counter), and interacts with several processes (see figure 5.1). The operation of a DQSM is partitioned into 6 states (below, we are considering the DQSM of priority 1 of the *forward* bus only):

`WaitSignals` It issues 5 requests:

1. to the `REQ_SET_1_RL` mailbox for a possible signal coming from the `Monitor` when `REQ_1` bit is set (`SlotReq`).
2. to the `ACCESS_Q_1_LR` mailbox for a possible signal coming from the QAP when a QA segment of priority 1 is ready for transmission (`CountDown`).
3. to the `PRI_REQ_1_RL` mailbox for a possible signal coming from the `Monitor` when `REQ_2` bit is set (`IncCntrs`).
4. to the `SELF_PRI_1_LR` mailbox for possible signal coming from the QAP when a QA segment of priority 2 is ready for transmission (`IncCntrs`).
5. to the `EMPTY_SLOT_1_LR` mailbox for possible signal coming from the `Monitor` which an empty slot is encountered (`EmptySlot`).

`SlotReq` The `REQ_cntr` is incremented by one.

`CountDown` The process sets the `CD_cntr` to the `REQ_cntr` before it signals its associated RQSM process to send a request. Note that the operation of RQSM and DQSM is independent of each other, which means that a QA segment is permitted to be transmitted before the request is sent.

`IncCntrs` It increments the `REQ_cntr` when it is in idle state; otherwise, it increments the `CD_cntr`.

`EmptySlot` It decrements the `REQ_cntr` when it is in idle state; otherwise, it decrements the `CD_cntr`. If the `CD_cntr` is zero, it sets the BUSY bit to `YES` and advances to state `PermitXtm`.

PermitXtm It sends the **TX_BUS_1_LR** signal to the **Transmitter** process.

In state **WaitSignals**, an arrival of a signal from any process wakes up the DQSM_1 and it proceeds to the state enclosed in the bracket. At other states, control returns to state **WaitSignals** except when control is in state **EmptySlot** and the **CD_cntr** is zero.

5.5.6 Request Queue State Machine

The function of the RQSM is to send a request for a QA segment generated within the station. There are six RQSMs (*buses* × *priorities*) within the station and each RQSM_I of the *forward* (*reverse*) bus is associated with a DQSM_I of the *reverse* (*forward*) bus. Every RQSM maintains a **REQM_Q_cntr** (request queue) counter which records the number of outstanding requests of its priority level. The operation of a RQSM_I process starts in state **WaitReqs**. Then, it reacts to two signals:

- A **LOCAL_REQ** signal from its associated DQSM_I (about a self-generated QA segment) prompts the process to increment the **REQM_Q_cntr**.
- A **REQ_NOT_SET** signal from the **Monitor** triggers the process to go state **CheckReq**. There, it returns to state **WaitReqs** if the **REQM_Q_cntr** is zero; otherwise, it decrements the **REQM_Q_cntr** and sets the **REQ_I** to **YES**.

Having reacted to the signal, the RQSM_I returns to state **WaitReqs**.

5.5.7 Transmitter process

There are six **Transmitter** processes (*buses* × *priorities*) within a station. Three **Transmitter** processes of the *forward* (*reverse*) bus are associated with three DQSMs of the *forward* (*reverse*) bus. The **Transmitter_I** transmits QA segments of priority

I via the `PortLR` (`PortRL`) of the station onto the *forward* (*reverse*) bus. The process waits for a signal from its associated DQSM in state `WaitTransmit`. When a permission is granted by the DQSM, it waits for the `EOT` (End of Transmission) event to be activated by the ACF before starting to transmit the QA segment stored in its buffer (this is why a busy slot triggers two `BOT`s whereas an empty slot triggers one). Having completed its transmission, control goes to state `EPacket`. Here, it invokes the `stop` operation and clears its buffer. Then, it notifies the `Monitor` about the completion of a QA segment transmission and signals the QAP to fetch another QA segment.

5.5.8 Queued Arbitrated Receive process

The QAR process identifies the QA segment designated to the current station and creates multiple Reassembly State Machines (RSMs) to handle concurrent receipt of multiple IMPDUs addressed to the station. Each RSM will administer one IMPDU. We assumed that all the packets are delivered in order and the packets are not corrupted during the transmission. The attribute `MIDlist` comprises the MIDs of all the IMPDUs that are being processed by the RSMs. The QAR waits for a possible `RX_BUS` signal from the `Monitor` in state `WaitPacket`. Having notified by the `Monitor` about an arriving QA segment, it wakes up in state `CheckPktStatus`. A RSM process is only created when a QA segment is addressed to this station, the VCI value matches the value which the QAR is programmed to handle, and the MID is not on the `MIDlist`. Otherwise, the QA segment is ignored and control returns to state `WaitPacket`. In the case where the QA segment is designated to this station, the QAR informs the proper RSM about the reception of a QA segment and then advances to state `EndPacket` to wait for the `EMP` (End of My Packet) event. Having received the entire packet, it informs the Monitor to set the PSR bit on the subsequent ACF and returns to the state `WaitPacket`.

5.5.9 Reassembly State Machine

The purpose of the RSM is to reassemble all the DMPDUs derived from an IMPDU (at the source station) into the original IMPDU. This process terminates after the entire IMPDU is reassembled. When the RSM is created, it enters state `BOMPktArr` and saves the beginning-end tag value, the transmit sequence number and the length of the MSDU of the first DMPDU. Then it advances to state `NextPktArr` and requests the parent process (the QAR process) to inform him when its next QA segment arrives. A signal from the QAR restarts the process in state `CheckPktStatus`. Since we are interested only in the entire IMPDU being properly received and not the contents of the IMPDU, the RSM compares the transmit sequence number and only accumulates the length of the MSDU in each subsequent DMPDU. When the entire IMPDU is received (Segment Type = EOM), control goes to state `Validation` and performs the two-step verification scheme described in section 3.5.3. Having received the entire IMPDU, the process terminates itself in state `stop`.

5.6 Summary

We have presented the indepth details of implementing a DQDB protocol in `smurph`. Several aspects are essential when specifying the protocol.

1. Whenever a process is awakened by a channel event, a delay is necessary to ensure that the event has been removed by the system. Otherwise, the process would operate on the same event endlessly (presence of livelocks).

2. When a state transition occurs within a process or a station (from process to process), the address of an environmental attribute (e.g. `ThePacket`) has to be saved if the object pointed to by the variable is to be referenced later.

3. If the message and packet formats are extended, it is required to include `UMessage` and the packet type (e.g. `QASegment`) when creating a traffic pattern. For example:

```
traffic UTraffic (UMessage, QASegment) {
   ... additional attributes ...
};
```

4. The environmental variable `NStations` (indicates the number of stations presented in the network) should not be referenced before or during the configuration phase since it is still being updated.

5. The process synchronization is facilitated by mailboxes. However, they are found to be inconvenient when a parent process wishes to communicate with one of its child processes (the total number of the children is known only during the course of the simulation) as mailboxes can not be established dynamically. The problem is resolved by implementing a `signal` command which permits the parent process to send a signal to its child process(es) without referring to a mailbox.

The presented implementation has illustrated the fundamental concepts and facilities of **smurph**. The formalism of **smurph** requires a manual translation of protocol specifications into a set of extended finite state machines. A communication protocol described by low level specification languages such as **smurph** maps closely to the hardware details. In effect, **smurph** specification can be written in a one to one correspondence with the real network. Thus, porting a **smurph** specification to the physical world would require minor modifications. On the other hand, **smurph** specifications lack formally defined semantics. Since communication protocol are usually defined informally or in a formalism different from **smurph**, translating these specifications to **smurph** specifications may neglect some of the protocol important features. The issue is how can we ensure that a **smurph** specification has all the required service properties of the original protocol specification? We show how **smurph** observers can be used for that purpose.

In our case study, three DQDB networks with 16, 32 and 48 stations have been considered. Stations are uniformly distributed along the bus, and have local clock errors with a maximum value of 0.0002 per 1000 ITU. Each station sends and receives messages with equal probability. The transmission rate for every station is the same, which in our case is 1000 ITU/bit for a 150 Mbps DQDB. The traffic density ranges from overload to underload. The message limit is set to 50,000 in every simulation experiment.

Chapter 6

Conformance Testing of DQDB by Observers

6.1 Introduction

There are two drawbacks in systematically probing a given protocol implementation for its correctness using a series of test suites and monitoring the outputs. First, without any knowledge about the internal structure of the protocol implementation, the approach may not be convincing as the inputs usually come from an infinite set. Second, it gives no guarantee that the protocol implementation is free of deadlocks or livelocks. We discuss some partial resolutions of these drawbacks in the context of random state exploration.

A sound communication protocol must fulfill the basic properties: no deadlocks, no livelocks and no message loss. In **smurph**, the violation of these properties can be detected by examining the system output parameters:

- The simulation experiment stage (e.g. the maximum number of messages to be received).
- The global and local throughputs.
- The pending events queues.
- The events processed queues.
- The number of transmitted and received messages.

In addition to the above statistics, `smurph` also checks illegal operations such as acquiring a new packet with an occupied buffer using the `getPacket` command, transmitting multiple packets at the same port simultaneously and depositing a signal into a full mailbox. However, these features cannot guarantee that the service properties of the protocol are met because the internal functioning of the protocol (which involve several processes) cannot be checked on the basis of the above information and features. Instead, the `smurph` tools called *observers* are used. These observers are programmed by the users.

In this chapter, we describe how `smurph` observers are used to validate the behaviour of the DQDB protocol by random state exploration. The proposed DQDB draft [14] mentions the following service requirements which a DQDB implementation must support:

General Properties

- Slot delay before transmitting should be within the slots limit (see section 6.3.1) permitted by DQDB whenever a station has a QA segment to transmit—Progress property.
- The assembly and reassembly of IMPDUs.

Internal Properties

- Within a station, QA segments of higher priority should gain access to the bus ahead of QA segments of lower priority.
- If a request associated with a QA segment has not been sent on the *reverse* bus, access to the bus is not inhibited.
- The values of the CountDown counters and the Request counters are properly maintained.

These required service properties have been tested using **smurph** observers, except the reassembly of IMPDU property (validated by the Reassembly State Machine see section 5.5.9). Note that part of the progress property is the absence of locks. The **smurph** observer model is discussed in section 6.2. Section 6.3 describes the implementation of DQDB observers and then summarizes the results of the experiment obtained by the DQDB observers. The summary of the chapter is presented in section 6.4.

6.2 SMURPH Observers

Observers are tools provided by **smurph** to assert the service specifications of a protocol. Unlike any other random state exploration techniques, **smurph** observers have the ability to investigate both the external behaviour and the internal structure of a protocol implementation without changing the behaviour. The operation of observers and protocol processes is independent of each other.

Generally, each service property is administered by a single observer. In the case of multiple service properties, several observers are desired. Resembling regular processes, observers are defined by extended finite state machines. Unlike regular processes, observers are not affiliated with stations. Normally, **smurph** observers analyze the data structures belonging to a station whose process has just accomplished its mission and is returning to sleep.

General events that trigger the regular processes are not perceived by observers. Instead, observers are activated by two *meta-events*: a station process' transition and the passing of time. The first meta-event occurs when a station process arrives at a specific state which is being monitored by an observer. When the station process is still executing, the observer is unable to determine the behaviour of the process since the transaction of the process is atomic. In fact, `smurph` wakes up any observer requesting to scrutinize the current state of the process right after the process has exhausted the list of statements in its present state and has put itself to rest. The function by which observers request to monitor state changes of regular processes is `inspect` (id, pt, pn, st, cs). The functionality of the `inspect` operation resembles the `wait` command invoked by a regular process. The `inspect` operation takes 5 arguments, and their roles are as follows:

id It identifies the station(s) being observed. This parameter can be a station id, a station object pointer or `ANY`. If the symbol `ANY` is employed here, it implies that all the stations in the network are being monitored.

pt It indicates the type of the process.

pn It identifies the *nickname* of the process. Its purpose is to differentiate between processes derived from the same type. This must be used when the process type is inadequate to identify the process under observation.

st It defines the state of the process being monitored.

cs It is the observer state which the observer wishes to be in upon its return from a rest.

The `inspect` request can be interpreted as a declaration that the observer wants to remain suspended until the simulator awakes a regular process matching the parameters of the `inspect` command. When this happens, the observer is awaken in the indicated state and performs a validation procedure.

The second meta-event emerges when the observer suspends itself and requests to be waken up after a certain delay. This is done by invoking the command, `time_out`. This function acts as an alarm clock, and accepts two parameters:

- The first is the number of ITUs that the observer wants to sleep.
- The second specifies the state which the observer wishes to be in once it wakes up.

Since simulation time does not flow while the observer is performing its task, it acquires the relevant information instantly. After the observer completes its assignment, it sets the new awakening conditions before returning the control to `smurph`. This interrupt has no effect on the simulation.

The root process creates observers by invoking the `create` function. In `smurph`, it is considered illegal to create observers after the simulation has started. As a result, observers are created together with the station processes during the protocol initialization phase. The skeleton of the *observer* type is displayed below.

```
observer ps_obs  {

               ... local variables ...
               void setup (...) { ... };

               states {s0, s1, s2, ..., sn};

               perform {
                  state s0 :
                              ... specify waking conditions ...
                              ... suspends itself ...
```

```
                state s1 :
                           ... start activities ..
                           ... resume (s3);
                ...
                state sn :
                           ...
             };
};
```

`ps_obs` is the name of the newly derived `Observer` type. The local variables, a `setup` method, and a `perform` method are the 3 main components of the `Observer` type. All local variables are initialized in the `setup` method before observer proceeds to state `s0`. In the state `s0`, an observer can either suspend itself (by invoking the `inspect` or `timeout` command) or advance to another state. The `resume`(state) function allows the observers to move from one state to another state. It accepts only one argument, which is one of the process state identifiers. The `perform` method defines an extended state machine. Each state in the `perform` method corresponds to a state in the machine. To terminate an observer, the `terminate` command is issued.

Since simulation in `smurph` is driven by the arriving message from outside (`Client` *AI*) and there is no standard communication between the observers and the `Client` *AI*, we cannot force a process to switch from one state to another state. Nevertheless, it is possible to expose the most probable behaviour of the protocol by adjusting the distribution parameters of the traffic patterns.

6.3 DQDB observers

This section presents the `smurph` observers developed to validate the DQDB implementation, and the test results obtained during the simulation experiment. The service properties tested by `smurph` observers are: *slot delay*, *priority traffics*, *request slot*, and *slot counters*. The slot delay property ensures the system is progressing, and thus absence of locks. The rest of the properties focus on the internal structure of a station. Each service property was governed by a separate observer; as such *four* observers: `validate_SlotDelay`, `validate_Priority`, `validate_ReqBit`, and `validate_DQSM` were created. Furthermore, `validate_MCF`, `validate_QAP`, `validate_QAR`, and `validate_RQSM` were implemented to corroborate the internal consistency of each protocol process (see section 5.5). Internal consistency validation is done by comparing the expected responses of the protocol process from the view point of observers against the actual outcomes. Any contradiction causes the observers to print an error diagnosis. In effect, the idea applied here is similar to the redundant principle [2]. All the observers implemented were local observers except the `validate_SlotDelay` observer, which was a global (distributed) observer. Though most of the DQDB observers were performing local tests, their abilities were not constrained. On the contrary, they were more likely to detect errors since every station kept the current state of the network for accessing the bus. In each simulation performed on the three networks (see chapter 5), all the observers mentioned above were running. The observers supervised the processes belonging to a single station at a time. Since the operation of the DQDB protocol is symmetric, the description of the observers mentions the *forward* bus only. Appendix B contains the source codes of the observers.

6.3.1 Slot Delay

The `validate_SlotDelay` observer checks whether the number of slots which a station has to wait before receiving its requested slot is bounded according to the following calculation:

Assume the following notation:

- Let the station identifier be `stat_id`.
- The total length of the bus is `bus_len`.
- The slot length is `slot_len`.
- The number of stations in the network is `NStations`.
- The number of outstanding requests is `CD_cntr`.
- The distance between the head station and the station `stat_id` in terms of slots is `n_slots`.

First, the distance between the head station and the station `stat_id` is calculated. Since the distance between any two neighboring station is the same, we have:

```
distance_bet_2_stations = bus_len / (NStations - 1)

distance_bet_head_obser = stat_id * distance_bet_2_stations

n_slots = distance_bet_head_obser / slot_len

see_request = n_slots + 1

slot_delay = see_request * 2 + CD_cntr
(Priority 2)

slot_delay = see_request * 2 + CD_cntr + 1 + stat_id
(Priority 1)
```

```
slot_delay = see_request * 2 + CD_cntr + 2 + stat_id * 2
(Priority 0)
```

The request of the station **stat_id** reaches the head station after (**n_slots** + 1) number of slots. The one, added to the variable **see_request**, is the slot which carries the request. With three levels of priority, the value of the **slot_delay** for each priority is different. The **slot_delay** for priority 2 is the sum of the outstanding requests (`CD_cntr`) of priority 2 and twice the total of **see_request**. For priority 1, the **slot_delay** has to take into consideration the potential priority 2 requests queued by the *upstream* stations. Similarly, the **slot_delay** of priority 0 has to include the possible requests (priority 1 and priority 2) registered by the *upstream* stations. To compute the **slot_delay** on the *reverse* bus, the **stat_id** is replaced by (**NStations** - **stat_id** - 1). The maximum **slot delay** is when all the *downstream* stations have reserved for a slot but their requests have not been served yet.

There is one instance of **validate_SlotDelay** observer at each station (see its code in Appendix B). Upon creation, the observer computes the value of **see_request** before it proceeds to state **resume**. Here, it waits until a DQSM enters the countdown state. When this happens, the observer advances to state **SetCntr**. Depending on the priority of the DQSM, the appropriate **slot_delay** is computed in accordance with the method presented above. Now, whenever the **Monitor** process encounters an ACF at its port, the observer wakes up in state **Passingslot** and increments the attribute **n_slots** by one. If the DQSM arrives at state **IncCntrs**, the observer increments the **slot_delay** in state **PriReq** to account for the priority request sent by the *downstream* stations. Once the associated **Transmitter** of the DQSM starts transmitting in state **XPacket**, the observer arrives at state **Verify**, and compares the value of `n_slots` against the `slot_delay`. If `n_slots` exceeds the allowed limit, a short diagnosis is printed by the observer before returning to state `resume`. Note that the observer begins recording the passing slots after the DQSM has entered countdown state not after the request has been sent. This is because the sending of a request and the transmitting of a QA segment are two separate issues according to the second service property.

A problem was encountered when implementing the `validate_SlotDelay` observer. At the beginning of the simulation, the observer detected violations of the limit. Further analysis showed that it took approximately 248 slot times (due to propagation delay) for the first ACF with or without a request to reach the end station of a bus after the simulation had started. During this time, the head stations were hogging onto the slots since they have not seen any request; whereas the stations, situated in between the two head stations, which had entered the countdown state were unable to reserve the slots as they either have not seen any ACF or their requests are on the way to the end station. This problem was resolved by extending `smurph` with 3 functions: `suspend`, `resume` and `resetSPF`. The `suspend` command allows us to suspend the traffic generators from producing messages while the network is still in configuration phase. Thus, no observer is activated as stations cannot enter countdown state without messages. After the buses were saturated with ACFs (about 300 slot times later), we invoked the `resume` command to reinstate the suspended traffic generators. Now, a station which enters countdown state can request an empty slot without any difficulty. The `resetSPF` operation is usually issued right after the `resume` function is called. It resets all the statistical measurements collected during the initialization phase, and thus provides more precise performance measure.

The test results gathered during the simulation showed that QA segments of priority 2 gained access to bus within the bounded limit in any situation. Nevertheless, it was not the case for QA segments of priority 1 and priority 0 when the network was heavily overload. To elucidate why this scenario took place, assume that station *15* wishes to send a message (priority 0) to station *18*. First, station *15* requests for an empty slot on the *forward* bus. When the requested empty slot is on the way to station *15*, station *10* occupied it with QA segment of higher priority before sending a request. If this condition arises, the observer should update the `slot_delay` of station *15* since station *15* may have to use the empty slot acquired by station *10*. However, keeping track of which *upstream* stations utilize the empty slot does not help much as in this case there is no bound for lower priority slot delays. As a result, the verdict in this case was inconclusive unless a better solution could be found. Under heavy, medium, and light load, no errors were detected by the observer.

6.3.2 Priority Traffic Patterns

To conform the *priority traffic* property, the `validate_Priority` observer spies on all the DQSMs within a station and notes down any DQSM which enters the countdown state in state `EnterCD` (see its code in Appendix B). Whenever a `Transmitter` process is granted the permission to convey (state `WaitTransmit`), the observer proceeds to state `Verify` and checks whether the QA segment is to be stored in this buffer (a QA segment of priority I should be deposited in `Buffer_I_LR`) and to be send on the *forward* bus. The observer also investigates if any of the DQSMs of higher priority is still in the countdown mode (recorded in state `EnterCD`). Any violation causes the observer to output an error message. No violation was discovered by the observer.

6.3.3 Slot Request

To certify the *slot request* property, the `validate_ReqBit` observer monitors all the DQSM processes. Whenever the DQSM_I arrives at state `CountDown`, the observer advances to state `ReqOrXtm`. Here, it waits for two events:

- The RQSM_I process to enter state `ReqBitSet`-issues a request.
- The `Transmitter_I` process to attains at state `XPacket`-begins transmission.

For a violation to occur, the RQSM_I process has to issue a request for every QA segment in any situation (even if there are unrequested empty slots passing by) before the transmission of the QA segment can take place. Note that this may not be a violation under heavy load as most slots are occupied. Any occurrence of a reverse order proves it otherwise. Based on the test results, majority of the QA segments were transmitted before a request has been made when the network is under low load. Under heavy or medium load, only a few stations which are located near the head station exhibit such a behaviour.

6.3.4 Validating Distributed Queued State Machine

The `validate_DQSM` observer acts as a redundant copy of the DQSM processes. It maintains a copy of REQ counter and CD counter for each priority. The operation of the observer is as follows:

- Whenever the `Monitor` process of the *reverse* bus returns from the `SlotHeader` state, the observer proceeds to state `ACFStatus`. Subsequently, it examines the status of the request field and updates its counters accordingly.
- Whenever the QAP process proceeds to the `MovetoXtmBuf` state, the observer wakes up at state `SelfReq` and increments its appropriate counters if necessary.

- Whenever a DQSM enters state **`CountDown`**, the observer arrives at state **`SetCD`** and operates on its CD counter.
- Whenever a DQSM process arrives at **`EmptySlot`** state, the observer advances to state **`Verify`**, and compares the values of its REQ and CD counters against the values of the REQ and CD counters of the DQSM. Unexpected values cause the observer to print out an error disgnostics.

Recall that in **`smurph`**, if multiple events awaited by a process occur in the same ITU, the order of processing the events is non-deterministic. It is possible that the ACF on both buses can arrive at a station simultaneously. In such case, the operation to be performed on the REQ and CD counters can be unpredictable. To illustrate, assume that the slot encountered on the *forward* bus is empty, each request subfield of the ACF on the *reverse* bus is set, and the DQSM_I is in idle state and its REQ counter is 5. Then, the DQSM_I randomly performs one of the actions listed below:

- enters state **`SlotReq`** to increment its REQ counter–priority 0 request.
- enters state **`IncCntrs`** to updates its REQ counter–higher priority requests.
- enters state **`EmptySlot`** to decrement its REQ counter–an empty slot.

If the decrement precedes the increment, the value of the REQ counter of the observers is different from the value of the REQ counter of the DQSM_I when the validation procedure is carried out in state **`Verify`** since the observer has already accounted the requests on the *reverse* bus. Similar situation can occur when the DQSM_I is in countdown state. To overcome the problem, the observed state **`EmptySlot`** must be reached the last. Thus, a change in **`smurph`** was required. A priority **`wait(`***events*, *state*, *pri*) command was incorporated into **`smurph`** by P. Gburzyński. Unlike regular **`wait`** operation, it accepts 3 arguments. The first two arguments serve the same purpose as in a regular **`wait`**. The last argument, which is optional, makes it possible to arrange the preference order of restarting a process if multiple events awaited by the process occur within the same ITU. The *pri* argument of the smallest value is processed ahead of the others. In our case, the lowest priority was assigned to state `EmptySlot` of the DQSM process. When the control proceeds to the `EmptySlot`, all the counters would have been incremented. Hence, the observer can confirm the action of DQSM. This alteration has no effect on the actual simulation. According to the results gathered during the simulation, the operation of the DQSM was correctly maintained.

6.3.5 Validating MAC Convergence Function

The `validate_MCF` observer is concerned with setting the attributes of the DMPDU (Derived MAC Protocol Data Unit), and maintaining the operation of the `Tx_Seq_Num` (Transmit Sequence Number) counter and the `BEtag` (Beginning-End tag) counter of each priority. It can be perceived as a carbon copy of the MCF process. For each priority, the observer has three attributes: `Tx_Seq_Num` counter, `BEtag` counter and `Msg_len`. The variable `Msg_len` accumulates the MSDU (MAC Service Data Unit) length carried in each DMPDU. Initially, all the attributes are set to zero. When the MCF process arrives at state `FDMPDU`, the observer saves the MSDU length of the current DMPDU in `Msg_len` and also ratifies the value coded in the `BEtag` subfield. The validation scheme takes place whenever the MCF process makes a transition to state `WaitAccess`. Here, the attributes such as MID, segment type, DA (Destination Address), and SA (Source Address) are revealed. If the segment is the last DMPDU (EOM code), the observer compares the `BEtag` value of this DMPDU against the first DMPDU (recorded). A mismatch causes the observer to print an error message.

The observer verified that the attributes of each DMPDU were set to the appropriate values in the simulation.

6.3.6 Validating Queued Arbitrated Portion process

The operation of the `validate_QAT` observer is simple and straightforward. The attributes, including the MID (Message IDentifier), the VCI (Virtual Channel Identifier) and Sequence Number, of a DMPDU (Derived MAC Protocol Data Unit) are duplicated by the observer in the MCF process. Whenever the QAP process fetches a new DMPDU, the observer analyzes the status of the DMPDU with those recorded in the MCF process. A mismatch would indicate either the DMPDU is out of sequence or the implementation of the QAP process is at fault. In the simulation, the QAP process received all the DMPDUs passed to it by the MCF process orderly.

6.3.7 Validating Queued Arbitrated Received process

The `validate_QAR` observer examines the consistency of the QAR process. To ensure that the `Monitor` process and the QAR process are looking at the same slot, the observer collects the relevant information when the `Monitor` proceeds to state `SlotBody`. These data are then collated with those perceived by the QAR process in state `PktArr`. In addition, the observer makes sure that the QA segment is addressed to the current station and the MID is not presently handled by an RSM (Reassembly State Machine) whenever the QAR process arrives at state `BOMPktArr`. The observer confirmed that the QAR process was correctly implemented.

6.3.8 Request Queued State Machine Observer

The basic function of the `validate_RQSM` observer is to attest the operation of the RQSM_I process. Additional processes participate in the validation procedure are: DQSM_I and `Monitor` of the *reverse* bus. Both DQSM_I and `Monitor` processes interact with the RQSM_I by providing the inputs (see figure 5.1), and the observer anticipates the responses. The observer retains a copy of the `REQ_Q_I` (Request Queue) counter. Whenever the DQSM_I is in `CountDown` state and acquaints the RQSM_I to send a request, the observer increments its `REQ_Q_I` counter. Whenever the `Monitor` attains at state `SlotHeader`, the observer inspects the status of the REQ_I bit. In the case where the REQ_I bit is not set, the `REQ_Q_I` is decremented if it is > zero. To ensure that the `REQ_Q_I` counter of the RQSM_I has the correct value, the observer compares the value of the two counters before the RQSM_I returns to state `WaitReqs` from state `SetReqBit`. However, these values may differ if priority wait is not employed here. This can only happen if the addition (notified by DQSM_I) and subtraction (awarded by `Monitor`) of the `REQ_Q_I` counter of the RQSM_I takes place in the same ITU. If the RQSM_I enters state `LocalReq` to increment its counter first, the two counters are equal; otherwise, they differ by one at most. Although one might argue that this bound is acceptable, others might not be convinced. To have absolute knowledge of the value of the `REQ_Q_I` counter, the state `SetReqBit` is given the lowest priority within the RQSM_I process. No errors were spotted by the observer.

6.4 Summary

We have presented some details of developing **smurph** observers to validate a DQDB implementation. Although there were no errors located by the observers, we came across two important issues in the validating process. First, when the network traffic is heavily loaded, a station may have to send a QA segment's request before it can transmit the QA segment since most slots are either reserved by the *downstream* stations or occupied by the *upstream* stations. This, however, may not be a violation of the *slot request* property. Second, the *slot counters* property was difficult to validate as the precise sequence of events (within the DQSM process) cannot be predicted. To deal with the difficulty, we proposed a priority wait function to be included in **smurph**. This function allows us to anticipate the choice of a restarting a protocol process that waits for several waking events. This modification has no effect on the actual simulation. Additional features such as **suspend**, **resume** and **resetSPF** were added to **smurph** to increase its efficiency. The **suspend** function permits the users to discontinue the message generating activity and to continue after the **resume** command is invoked. The **resetSPF** function is issued when the users want to ignore all the performance measures gathered thus far in the simulation. This operation is usually called after the network configuration phase.

When developing DQDB observers, the prominent issue is deciding what service properties should be tested since there are no existing conformance testing standard for the DQDB protocol. In another word, did the simulation experiment cover all the service specifications? Since some service specifications of the implementation are apparent (e.g. slot generator), the author felt that the service properties excerpted from the proposed DQDB draft were the most significant requirements of the protocol.

Chapter 7

Conclusions and Directions for Future Research

In this thesis, we have investigated the feasibility of conformance testing of low-level communication (MAC) protocols implemented in the **smurph** modeling environment. Our main interest is in developing and implementing **smurph** observers to determine the behaviour of the protocols by random state exploration. The chosen MAC protocol is called Distributed Queue Dual Bus, a proposed standard protocol for MANs.

The starting point of our case study is the proposed DQDB standard draft [14], which gives an informal specification of the protocol (written in English with some mathematical notations), and implicitly defined the required service properties of the protocol. The transformation of the specification to an extended finite state machine model coded in C++ (**smurph** specification) was done manually. The derived model can exhibit the actual behaviour of the protocol when executed by the **smurph** run-time system, and thus can be scrutinized.

When running the DQDB implementation in **smurph**, the timely question to ask is: *Are the service properties required by the standard met*? With the complexity of the protocol, we did not attempt to perform a full state-space exploration, neither did we consider proving the properties. Instead, we have relied on random-state exploration since it is always feasible.

Some service properties, such as deadlocks or livelocks, were not tested explicitly, but their absence can be inferred from the standard statistics collected by **smurph** during the simulation experiment. To illustrate:

- The growing global throughputs contributed by all stations implies absence of livelocks.
- The growing local throughputs at each station implies no starvation.
- The non-empty event queues implies absence of deadlocks.

Nevertheless, observing throughputs is insufficient to confirm that the service properties consent with global or local assertions are met. In order to ensure that these service properties are fulfilled, we are required to use observers. These observers operate independently from protocol processes. In effect, protocol processes are unaware of the presence of observers. Although observers cannot react to regular events which activate protocol processes, they can perceive the state transitions of the protocol processes. This forces a careful design of protocol processes such that their state transitions facilitate the observation of their expected behaviour. Since observers are derived in a different format from protocol processes, they together result in a self-checking code.

In our case study, four observers were implemented to monitor the *four* informal service specifications: *slot delay*, *priority traffics*, *request slot* and *slot counters* respectively. Additional observers are implemented to test the consistency of station processes. Several obstacles were encountered during the course of implementing the observers. To overcome them, we proposed several functions: the new priority `wait`, `suspend`, `resume` and `resetSPF` to be incorporated into `smurph`. The new `wait` operation allows the users to schedule the order of awaited events for a restarting process while the process is in a suspend state. The `suspend` commands prevents a traffic pattern from generating messages while in network configuration phase, whereas the `resume` operation reinstates the suspended traffic pattern. The `resetSPF` command provides more precise performance measure by ignoring the statistical results during the initialization phase.

Extensive testing indicates that the protocol is correctly implemented with respect to the informal service specifications. Since the `smurph` implementation has been tested using observers (such observation would be very difficult to build in hardware), it may now serve as a detailed specification for hardware implementation of the protocol. This process not only is cost effective but also yields significant confidence in the design.

Based on our experience, we postulate that observers be allowed to perceive events that protocol processes respond to. However, to maintain their passive nature, they must not be permitted to initiate activities that can generate new events. With this modification, the expressive power of observers will increase and also the convenience of using them.

Given that testing an implementation in the `smurph` environment is by random state exploration, can the test results ensure the correctness of the entire protocol implementation? The degree to which the protocol has been validated is still unknown because of the nature of random state exploration technique. However, it is certain that the search space explored was beyond the ability of a human without machine assistance. An experiment by West [38] to validate the OSI Session Layer using both exhaustive and random state exploration techniques shows that as the complexity of a protocol increases, the size of the state space is such that exhaustive exploration becomes highly redundant, and a random exploration of a subset of the state space becomes an effective method of detecting errors. We fully support this observation.

To conclude, **smurph** proved to be a valuable tool in modeling a low-level communication protocol in a realistic executable environment. The test results obtained from a simulation experiment can concede substantial confidence in the quality of the implemented protocol even though they do not insinuate the correctness of all protocol states.

Bibliography

[1] J. M. Ayache, P. Azema, and M. Diaz. Observer: a concept for on-line detection for control errors in concurrent systems. *9th Int. Symp. FTC, Madison*, pages 1–8, June 1979.

[2] J. M. Ayache, R. Molva, and M. Diaz. Observer: A run-time checking tool for LANs. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification V*, pages 495–506, 1985.

[3] M. J. Berard. Developing CSMA/CD Protocols with LANSF Observers. Master's thesis, University of Alberta, 1991.

[4] S. Budkowski and P. Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, pages 3–2314, 1987.

[5] J. P. Courtiat. Estelle: A powerful dialect of Estelle for OSI Protocol Description. In *Protocol Specification, testing, and verification*, 1988.

[6] M. W. Garrett and S. Q. Li. A Study of Slot Reuse in Dual Bus Multiple Access Networks. *IEEE Infocom*, pages 617–629, 1990.

[7] P. Gburzyński and P. Rudnicki. A better-than-Token protocol with bounded packet delay time for Ethernet-type LAN's. In *Symposium on the Simulation of Computer Networks*, pages 110–117, Colorado Springs, Co., August 1987. IEEE.

[8] P. Gburzyński and P. Rudnicki. Using time to synchronize a Token Ethernet. In *Proceedings of CIPS Edmonton*, pages 280–288. Canadian Information Processing Society, November 1987.

[9] P. Gburzyński and P. Rudnicki. A note on the performance of ENET II. *IEEE Journal on Selected Areas in Communications*, 7:424–427, April 1989.

[10] P. Gburzyński and P. Rudnicki. A virtual token protocol for bus networks; correctness and performance. *INFOR*, 27:183–205, 1989.

[11] P. Gburzyński and P. Rudnicki. The SMURPH Protocol Modeling Environment. Technical report, University of Alberta, 1992.

[12] P. Gburzyński, P. Rudnicki, and W. Dobosiewicz. An Ethernet-like CSMA/CD protocol for high speed bus LANs. *IEEE INFOCOM*, pages 238–245, 1990.

[13] G. J. Holzmann. *Design and Validation of Computer Protocols*. Penrice Hall softwares series, 1991.

[14] IEEE. Project 802 - Local and Metropolitan Area Networks, Proposed Standard: Distributed Queue Dual Bus (DQDB) Subnetwork of a Metropolitan Area Network (MAN), October 1990.

[15] J. O. Limb and C. Flores. Description of Fasnet - a unidirectional Local Area Communication Networks. *Bell Systems Technical Journal*, 61(7):1413–1440, September 1982.

[16] F.J. Lin, P.M. Chu, and M.T. Liu. Protocol Verification Using Reachability Analysis, The State Explosion Problem and Relief Strategies. *ACM SIGCOMM*, 17(5):215–274, 1987.

[17] R. J. Linn. Conformance Testing for OSI Protocols. *Computer Networks and ISDN Systems*, 18:203–219, 1990.

[18] N. F. Maxemchuk and K. Sabnani. Probabilistic Verifcation of Communication Protocols. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification VII*, pages 307–320, 1987.

[19] J. F. Mollenauer. Metropolitan Area Networks: A New Application for Fiber. *Photonics Spectra*, pages 159–161, March 1990.

[20] R. M. Newman, Z. L. Budrikis, J. L. Hullet, D. Economou, F. M. Fozdar, and R. D. Jeffery. QPSX: A Queued Packet and Synchronous Circuit Exchange. In *Proc. 8th Int. Conf. Comp. Comm.*, pages 288–293, Munich, 1986.

[21] R. M. Newman, Z. L. Budrikis, and J. L. Hullett. The QPSX MAN. *IEEE Communication Magazine*, 26(4):20–28, April 1988.

[22] R. M. Newman and J. L. Hullett. Distributed Queueing: A fast and efficient packet access protocol for QPSX. In *Proc. 8th Int. Conf. Comp. Comm.*, pages 294–299, Munich, 1986.

[23] F. Orava. Formal Semantics of SDL specifications. In *Proc. IFIP Symposium of Protocol Specification, Testing, and Verification VIII*, 1988.

[24] B. Pehrson. Protocol Verification for OSI. *Computer Networks and ISDN Systems*, 18:185–201, 1990.

[25] D. Rayner. Towards standardized OSI conformance tests. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification V*, pages 441–460, 1985.

[26] D. Rayner. OSI Conformance Testing. *Computer Networks and ISDN Systems*, 14:79–98, 1987.

[27] M. A. Rodrigues. Erasure Node: Performance Improvements for the IEEE 802.6 MAN. *IEEE Infocom*, pages 636–643, 1990.

[28] B. Sarikaya, G. v. Bochmann, and E. Cerny. A Test Design Methodology for Protocol Testing. *IEEE Trans. Communications*, 13(4):389–395, 1987.

[29] Y. N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple UIO sequences. In *Protocol Specification, Testing, and Verification*, pages 131–143, 1989.

[30] G. v. Bochmann. Usage of protocol development tools: The results of a survey. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification VII*, pages 139–161, 1987.

[31] G. v. Bochmann. Protocol Specification for OSI. *Computer Networks and ISDN Systems*, 18:167–184, 1990.

[32] G. v. Bochmann and M. Deslauriers. Combining ASN1 support with the LOTOS languages. In *Proc. IFIP Symposium on Protocol Specification, testing, and verification IX*, 1989.

[33] G. v. Bochmann and R. Dssouli. Error detection with multiple observers. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification V*, pages 483–494, 1985.

[34] G. v. Bochmann and R. Dssouli. Conformance Testing with Multiple Observers. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification V*, pages 217–229, 1986.

[35] G. v. Bochmann and J. R. Zhao. Reduced Reachability Analysis of Communication Protocols: A New Approach. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification VI*, pages 243–254, 1986.

[36] B. Wang and D. Hutchinson. Protocol Testing Techniques. *Computer Communications*, 10(2):79–87, April 1987.

[37] C. H. West. An automated technique of communications protocol validation. *IEEE Trasaction on communications*, 26(8):1271–1275, 1978.

[38] C. H. West. Protocol Validation by Random State Exploration. *Proc. IFIP Symposium Protocol Specification, Testing, and Verification VI*, pages 233–242, 1986.

[39] C. H. West and P. Zafiropulo. Automated validation of a communications protocol: the CCITT X.21 recommendation. *IBM Journal Res. Develop.*, 22(1), January 1978.

[40] P. Zafiropulo. Protocol validation by Duologue Matrix analysis. *IEEE Transaction on communications*, 26(8):1187–1194, 1978.

[41] M. Zukerman and P. G. Potter. A Protocol for Eraser Node Implementation within the DQDB Framework. *Telecom Australia Report*, pages 1400–1404, 1990.

Appendix A

The Implementation of DQDB

A.1 Unsynchronized Slot Generator

```
void  Slotter::setup (int direction) {
   S->SlHeader.TP = SLOT;
   S->SlHeader.BUSY = S->SlHeader.SLOT_TYPE = S->SlHeader.PSR = NO;
   S->SlHeader.REQ_0 = S->SlHeader.REQ_1 = S->SlHeader.REQ_2 = 0;
   S->SlHeader.ILength = S->SlHeader.TLength  = S->SlHdrLngth;
   S->SlHeader.Sender  = S->SlHeader.Receiver = NONE;
   setFlag (S->SlHeader.Flags, PF_full);
   if (direction == RIGHT)
      MyPort = S->PortLR;
   else
      MyPort = S->PortRL;
};

Slotter::perform {

   state StartSlot:
      MyPort->transmit (S->SlHeader, SlotDelay);

   state SlotDelay:
      MyPort->stop ();
      Slot_time = ((SlotLngth - 1) * ITUinBIT) + toss (max_time);
      Timer->wait (Slot_time, StartSlot);
};
```

A.2 MAC Convergence Function Block

```
#define   WAITEVENTS(ws0, ws1, ws2, ev0, ev1, ev2, st0, st1, st2) {
```

```
          ws0->wait(ev0, st0);
          ws1->wait(ev1, st1);
          ws2->wait(ev2, st2);
          }

#define   MCF_PRI(tp, macb) {
          MCFBuffer = &(macb)
          if (!tp->getPacket(MCFBuffer,FPacLen,FPacLen,FPHdrLen))
              proceed (WaitStates);
          else
              proceed (FDMPDU);
          }

#define   MCF_FSeg(msg, tag, Tx, ms) {
          msg = TheMessage->Length + MCFBuffer->ILength;
          MCFBuffer->COM_PDU_Hdr.BAsize = MCP_Len + msg;
          MCFBuffer->COM_PDU_Hdr.BEtag = tag;
          MCFBuffer->DM_Hdr.Seq_Num = Tx;
          ms = FDMPDU;
          }

#define   MCF_DONE(tp, macb, Tx, qs, ms, tag, st) {
          Tx = (Tx + 1) % 16;
          qs = NON;
          clearFlag(macb.Flags, PF_full);
          if (ms == LDMPDU) {
              tag = (tag + 1) % 256 ;
              proceed (st);
          } else {
              MCFBuffer = &macb;
              if (!tp->getPacket(MCFBuffer,MinLen,CPacLen,CPHdrLen))
                  proceed (WaitStates);
          } else
                  proceed (CDMPDU);
          }

MCF_Block::perform {

   state WaitIMPDUs:
      WAITEVENTS(U2TPattern, U1TPattern, UTPattern, ARRIVAL, ARRIVAL,
                 ARRIVAL, Prior2, Prior1, Prior0);

   state Prior2:
      MCF_PRI(U2TPattern,S->MacBuf_2);
```

```
state Prior1:
   MCF_PRI(U1TPattern,S->MacBuf_1);

state Prior0:
   MCF_PRI(UTPattern,S->MacBuf_0);

state FDMPDU: // Set COMMON PDU Header Info
   if (!GETPACKET(MCFBuffer, MinLen, FPacLen, FPHdrLen))
       proceed (WaitStates);
   else {
       MCFBuffer->COM_PDU_Hdr.RSVD = 0;
       switch (MCFBuffer->TP) {
         case 0 : MCF_FSeg(Msg_Len_0,BEtag_cntr_0,Tx_Seq_Num_0,Mstat_0);
                  break;
         case 1 : MCF_FSeg(Msg_Len_0,BEtag_cntr_0,Tx_Seq_Num_0,Mstat_0);
                  break;
         case 2 : MCF_FSeg(Msg_Len_0,BEtag_cntr_0,Tx_Seq_Num_0,Mstat_0);
                  break;
       }
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.DA = MCFBuffer->Receiver;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.SA = MCFBuffer->Sender;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.PI = 1;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.Pad_Len = 0;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.QOS_Delay = 0;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.QOS_Loss = 0;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.CRC_32 = 0;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.Hdr_Ext = 0;
       MCFBuffer->COM_PDU_Hdr.MCP_Hdr.Bridging = 0;
       MCFBuffer->DM_Trl.PayLoad_Len =
                 MCFBuffer->ILength + MCP_Len + PHdrLngth;
       MCFBuffer->DM_Trl.PayLoad_CRC = 0;
       if (MCFBuffer->isLast ()) {
           MCFBuffer->DM_Hdr.MID = 0; // Single Segment Message
           MCFBuffer->DM_Hdr.Seg_Type = SSM;
           proceed (LDMPDU);
       }
       MCFBuffer->DM_Hdr.MID = MCFBuffer->Sender+(MCFBuffer->TP*NumSt);
       MCFBuffer->DM_Hdr.Seg_Type = BOM;
       proceed (WaitAccess);
   }// else

state CDMPDU: // Continuation of Message
     // Set Derived PDU Trailer
   switch (MCFBuffer->TP) {
      case 0 : Mstat_0 = CDMPDU;
               MCFBuffer->DM_Hdr.Seq_Num = Tx_Seq_Num_0;
               break;
      case 1 : Mstat_1 = CDMPDU;
               MCFBuffer->DM_Hdr.Seq_Num = Tx_Seq_Num_1;
               break;
      case 2 : Mstat_2 = CDMPDU;
               MCFBuffer->DM_Hdr.Seq_Num = Tx_Seq_Num_2;
               break;
   }
   MCFBuffer->DM_Trl.PayLoad_Len = MCFBuffer->ILength;
   MCFBuffer->DM_Trl.PayLoad_CRC = 0;
   MCFBuffer->DM_Hdr.MID = MCFBuffer->Sender+(MCFBuffer->TP*NumSt);
   if (MCFBuffer->isLast()) { // Check for Last Seg.
       MCFBuffer->DM_Hdr.Seg_Type = EOM;
       proceed (LDMPDU);
   }
   MCFBuffer->DM_Hdr.Seg_Type = COM;
   proceed (WaitAccess);

state LDMPDU:
   switch (MCFBuffer->TP) { // Last Segment
      case 0 : Mstat_0 = LDMPDU;
               MCFBuffer->COM_PDU_Trl.BEtag = BEtag_cntr_0;
               MCFBuffer->COM_PDU_Trl.Length = MCP_Len + Msg_Len_0;
               break;
      case 1 : Mstat_1 = LDMPDU;
               MCFBuffer->COM_PDU_Trl.BEtag = BEtag_cntr_1;
               MCFBuffer->COM_PDU_Trl.Length = MCP_Len + Msg_Len_1;
               break;
      case 2 : Mstat_2 = LDMPDU;
               MCFBuffer->COM_PDU_Trl.BEtag = BEtag_cntr_2;
               MCFBuffer->COM_PDU_Trl.Length = MCP_Len + Msg_Len_2;
               break;
   }
   MCFBuffer->DM_Trl.PayLoad_Len += PHdrLngth ;
   MCFBuffer->COM_PDU_Trl.PAD = 0; // Set Common PDU Trailer
   MCFBuffer->COM_PDU_Trl.RSVD = 0;
   proceed (WaitAccess); // IMPDU Length without Hdr & Trl

state WaitAccess:
   S->VCI = MCF_VCI; // Pass VCI, PayLoad Type and etc
   S->PayLoad_Type = 0; // to Queue Arbitrated Function Block
```

```
    S->Segment_Priority = 0;
    S->Hdr_Check_Seq = 0;
    if (MCFBuffer->Receiver >= MCFBuffer->Sender) {
        switch (MCFBuffer->TP) {
          case 0 : if (S->Mb.ACC_0_LR->put () == REJECTED)
                       print ("\nSignal Rejected at MCF - Bus_0_A");
                   Qstat_0 = FILL;
                   break;
          case 1 : if (S->Mb.ACC_1_LR->put () == REJECTED)
                       print ("\nSignal Rejected at MCF - Bus_1_A");
                   Qstat_1 = FILL;
                   break;
          case 2 : if (S->Mb.ACC_2_LR->put () == REJECTED)
                       print ("\nSignal Rejected at MCF - Bus_2_A");
                   Qstat_2 = FILL;
                   break;
} // switch
   } else {
        switch (MCFBuffer->TP) {
          case 0 : if (S->Mb.ACC_0_RL->put () == REJECTED)
                       print ("\nSignal Rejected at MCF - Bus_0_B");
                   Qstat_0 = FILL;
                   break;
          case 1 : if (S->Mb.ACC_0_RL->put () == REJECTED)
                       print ("\nSignal Rejected at MCF - Bus_1_B");
                   Qstat_1 = FILL;
                   break;
           case 2 : if (S->Mb.ACC_2_RL->put () == REJECTED)
                        print ("\nSignal Rejected at MCF - Bus_2_B");
                    Qstat_2 = FILL;
                    break;
        }  // switch
   } // else
   proceed (WaitStates);

state WaitStates:
   sum =  Qstat_0 + Qstat_1 + Qstat_2;
   switch (sum) {
      case 0 :  proceed (WaitIMPDDUs);
                break;
      case 1 :  if (Qstat_0 == FILL)
                    WAITEVENTS(S->Mb.DONE_0, U2TPattern, U1TPattern, RECEIVE,
                           ARRIVAL, ARRIVAL, Done0, Prior2, Prior1);
                if (Qstat_1 == FILL)
```

```
                        WAITEVENTS(S->Mb.DONE_1, U2TPattern, UTPattern, RECEIVE,
                               ARRIVAL, ARRIVAL, Done1, Prior2, Prior0);
                    if (Qstat_1 == FILL) {
                        WAITEVENTS(S->Mb.DONE_2, U1TPattern, UTPattern, RECEIVE,
                               ARRIVAL, ARRIVAL, Done2, Prior1, Prior0);
                    break;
           case 2 : if ((Qstat_0 == FILL) && (Qstat_1 == FILL))
                         WAITEVENTS(S->Mb.DONE_0, S->Mb.DONE_1, U2TPattern,
                               RECEIVE, RECEIVE, ARRIVAL, Done0, Done1, Prior2);
                    if ((Qstat_0 == FILL) && (Qstat_2 == FILL))
                         WAITEVENTS(S->Mb.DONE_0, S->Mb.DONE_2, U1TPattern,
                               RECEIVE, RECEIVE, ARRIVAL, Done0, Done2, Prior1);
                    if ((Qstat_1 == FILL) && (Qstat_2 == FILL))
                         WAITEVENTS(S->Mb.DONE_1, S->Mb.DONE_2, UTPattern,
                         RECEIVE, RECEIVE, ARRIVAL, Done1, Done2, Prior0);
                    break;
           case 3 : WAITEVENTS(S->Mb.DONE_0, S->Mb.DONE_1, S->Mb.DONE_2, RECEIVE,
                    RECEIVE, RECEIVE, Done0, Done1, Done2);
                    break;
        } // switch

   state Done0:
      MCF_DONE(UTPattern,S->MacBuf_0,Tx_Seq_num_0,Qstat_0,Mstat_0,
               BEtag_cntr_0,Prior0);

   state Done1:
      MCF_DONE(U1TPattern,S->MacBuf_1,Tx_Seq_num_1,Qstat_1,Mstat_1,
               BEtag_cntr_1,Prior1);

   state Done2:
      MCF_DONE(U2TPattern,S->MacBuf_2,Tx_Seq_num_2,Qstat_2,Mstat_2,
               BEtag_cntr_2,Prior2);

};
```

A.3 Queued Arbitrated Portion

```
#define   QAFEVENTS(ws0, ws1, ws2, ev, st0, st1, st2) {
          ws0->wait(ev, st0);
          ws1->wait(ev, st1);
          ws2->wait(ev, st2);
          }
```

```
#define   ACC(qbuf, macb, donex, wt, accq, fin, pri) {
          if (qbuf->isFull()) {
              donex->erase();
              wt = donex;
              proceed(WaitStates);
          } else {
              MCFBuffer = &macb;
              QABuf = qbuf;
              ACCESS_Q =accq;
              DONE = fin;
              Priority =pri;
              proceed(MovetoXtmBuf);
              }
          }

QAP::perform {

   state WaitSignals:
      QAFEVENTS(ACC_2,ACC_1,ACC_0,RECEIVE,Access2,Access1,Access0);

   state Access0:
      QAFACC(QABuffer_0,S->MacBuf_0,DONE_X_0,wt_0,ACCESS_Q_0,
             S->Mb.DONE_0,Priority_0);

   state Access1:
      QAFACC(QABuffer_1,S->MacBuf_1,DONE_X_1,wt_1,ACCESS_Q_1,
             S->Mb.DONE_1,Priority_1);

   state Access2:
      QAFACC(QABuffer_2,S->MacBuf_2,DONE_X_2,wt_2,ACCESS_Q_2,
             S->Mb.DONE_2,Priority_2);

   state WaitStates:
      QAFEVENTS(wt_2,wt_1,wt_0,RECEIVE,Access2,Access1,Access0);

   state MovetoXtmBuf:
      MCFBuffer->VCI            = S->VCI;
      MCFBuffer->PayLoad_Type = S->PayLoad_Type;
      MCFBuffer->Seg_Prio       = S->Segment_Priority;
      MCFBuffer->Hdr_Chk        = S->Hdr_Check_Seq;
      *QABuf                    = *MCFBuffer;
      if (ACCESS_Q->put () == REJECTED)
          print (QABuf->TP,"\nSignal Rejected at QAF - Access_Q : ");
      switch (Priority) {
         case 0 : wt_0 = ACC_0;
                  break;
         case 1 : wt_1 = ACC_1;
                  if (SELF_PRI_R_1->put () == REJECTED)
                      print ("\nSignal Rejected at QAF - SELF_PRI_1");
                  break;
         case 2 : wt_2 = ACC_2;
                  if (SELF_PRI_R_1->put () == REJECTED)
                      print ("\nSignal Rejected at QAF - SELF_PRI_1");
                  if (SELF_PRI_R_2->put () == REJECTED)
                      print ("\nSignal Rejected at QAF - SELF_PRI_2");
                  break;
      } // switch
      if (DONE->put () == REJECTED)
          print ("\nSignal Rejected at QAF - Done");
      proceed (WaitStates);
};
```

A.4 Monitor

```
Monitor::perform {

   state WaitSlot:
      MyPort->wait (BOT, SlotCheck);

   state SlotStatus:
      if (ThePacket->TP == SLOT) {
          *SlotPtr = TheSlot;
          proceed (SlotHeader);
      }
      else
          proceed (SlotBody);

   state SlotHeader :
      if (!(*SlotPtr)->BUSY)  { // Slot is Unused
          if (EMPTY_SLOT_2->put () == REJECTED)
              print ("\nSignal Rejected at Mtr - Empty_Slot_2");
          if (EMPTY_SLOT_1->put () == REJECTED)
              print ("\nSignal Rejected at Mtr - Empty_Slot_1");
          if (EMPTY_SLOT_0->put () == REJECTED)
              print ("\nSignal Rejected at Mtr - Empty_Slot_0");
      }
      switch ((*SlotPtr)->REQ_2) {
        case  0 : if (REQ_NOT_SET_2->put () == REJECTED)
             print ("\nSignal Rejected at Mtr - Req_Not_2");
                  break;
        case  1 : if (REQ_SET_2->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Req_Set_1");
                  if (PRI_REQ_1->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Pri_Req_1");
                  if (PRI_REQ_2->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Pri_Req_2");
                  break;
      }
      switch ((*SlotPtr)->REQ_1) {
        case  0 : if (REQ_NOT_SET_1->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Req_Not_1");
                  break;
        case  1 : if (REQ_SET_1->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Req_Set_1");
                  if (PRI_REQ_1->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Pri_Req_1");
                  break;
      }
      switch ((*SlotPtr)->REQ_0) {
        case  0 : if (REQ_NOT_SET_0->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Req_Not_0");
                  break;
        case  1 : if (REQ_SET_0->put () == REJECTED)
                      print ("\nSignal Rejected at Mtr - Req_Set_0");
                  break;
      }
      if (REQ_SET_PSR->nonempty ()) {
          REQ_SET_PSR->erase ();
          (*SlotPtr)->PSR = YES ;
      }
      skiptio (WaitSlot);

  state SlotBody :
      *PktPtr = TheSegment; // Signal Check Pkt Status
      if (RX_BUS->put () == REJECTED)
          print ("\nSignal Rejected at Mtr - Rx_Bus");
      skipto (WaitSlot);
};
```

A.5 Distributed Queue State Machine

```
Dqsm::perform {

   state WaitSignals:
      REQ_SET->wait(RECEIVE, SlotReq);
      ACCESS_Q->wait(RECEIVE, CountDown, 3);
      PRI_REQ->wait(RECEIVE, IncCntrs);
      SELF_PRI_R->wait(RECEIVE, IncCntrs);
      EMPTY_SLOT->wait(RECEIVE, EmptySlot);

   state SlotReq:           // Update Request Cntr - Opposite
      if (REQ_cntr < Max_Cntr)
         (REQ_cntr)++;
      proceed (WaitSignals);

   state CountDown:
      if (Dqsm_State == COUNTDOWN)
         excptn ("Error in Dqsm at CountDown");
      CD_cntr  = REQ_cntr;
      REQ_cntr = 0;
      Dqsm_State = COUNTDOWN;
      if (LOCAL_REQ->put () == REJECTED)
         print ("\nSignal Rejected at Dqsm - Local_req");
      proceed (WaitSignals);

   state IncCntrs:
      if (Dqsm_State == IDLE) {
          if (REQ_cntr < Max_Cntr)
             (REQ_cntr)++;
      } else {
          if (CD_cntr < Max_Cntr)
             (CD_cntr)++;
      }
      proceed (WaitSignals);

   state EmptySlot:
     if (Dqsm_State == IDLE) {   // No Packet to sent
        if (REQ_cntr > 0)
           (REQ_cntr)--;
        proceed (WaitSignals);
     } else {                    // Waiting for Empty Slot
        if (CD_cntr > 0) {
           (CD_cntr)--;
```

```
            proceed (WaitSignals);
         } else
            proceed (PermitXtm);
      }  // End else
      proceed (WaitSignals);

   state PermitXtm :
      (*SlotPtr)->BUSY = YES;    // Signal for Transmit
      if (TX_BUS->put () == REJECTED)
         print ("\nSignal Rejected at Dqsm - Tx_Bus");
      Dqsm_State = IDLE;
      proceed (WaitSignals);
};
```

A.6 Request Queue State Machine

```
Rqsm::perform {
   state WaitReqs:
      LOCAL_REQ->wait(RECEIVE, LocalReq);
      REQ_NOT_SET->wait(RECEIVE, CheckReq, 1);

   state LocalReq:
      if (REQ_Q_cntr < Max_Cntr)
         (REQ_Q_cntr)++;      // Packet to be sent
      proceed (WaitReqs);

   state CheckReq:
      if (REQ_Q_cntr > 0)  // Request for Empty Slot
         proceed (SetReqBit);
      else
         proceed (WaitReqs);

   state SetReqBit:
      (REQ_Q_cntr)--;
      switch (Priority) {
        case 0 : (*SlotPtr)->REQ_0 = YES;
                 break;
        case 1 : (*SlotPtr)->REQ_1 = YES;
                 break;
        case 2 : (*SlotPtr)->REQ_2 = YES;
                 break;
      }
      proceed (WaitReqs);
};
```

A.7 Transmitter

```
Transmitter::perform {

   state WaitTransmit:   // Waiting for Permission
      TX_BUS->wait(RECEIVE, XPermitted);

   state XPermitted: // End of Slot Header
      MyPort->wait (EOT, XPacket);

   state XPacket: // Transmit QA Segment
      MyPort->transmit (*QABuffer, EPacket);

   state EPacket:
      MyPort->stop ();
      QABuffer->release (); // Signal Segment sent
      DONE_X->put ();
      proceed (WaitTransmit);
};
```

A.8 Queued Arbitrated Receive Block

```
QAR::perform  {

   state WaitPacket: // Received Packet
      RX_BUS->wait(RECEIVE, CheckPktStatus);

   state CheckPktStatus:
      if ((*QABuffer)->VCI != MCF_VCI) // Verified VCI
         excptn ("Unrecognized Virtual Channel Identifier");
      MID = (*QABuffer)->DM_Hdr.MID;
      if (((*QABuffer)->DM_Hdr.Seg_Type == BOM) ||
            ((*QABuffer)->DM_Hdr.Seg_Type == SSM)) {
         if ((*QABuffer)->COM_PDU_Hdr.MCP_Hdr.DA
               == TheStation->getId())
            proceed (BOMPktArr);
         else
            proceed (WaitPacket);
      } else { // Check COM Message
         if ((S->MIDList[MID]) == MID) {
            if (S->RcvPkt[MID]->signal () == REJECTED)
```

```
                   print ("\nSignal Rejected at QAF ");
               if ((*QABuffer)->DM_Hdr.Seg_Type == EOM) {
                   (S->MIDList[MID]) = -1;
                   (S->RcvPkt[MID])  = NULL;
               }
               proceed (EndPacket);
            } else {
               if (((*QABuffer)->Receiver == TheStation->getId ())
                   print("\n Error -> Missing BOM at QAF");
                   proceed (EndPacket);
               } else
                 proceed (WaitPacket);
            }  // else
       }  // end COM & EOM

   state BOMPktArr:   // Update Msg Iden. List
      if ((*QABuffer)->DM_Hdr.Seg_Type == BOM) {
         if ((S->MIDList[MID]) == -1) {
             (S->MIDList[MID]) = MID;
             S->RcvPkt[MID] = create RSM(Direction);
          }
      } else
             create RSM(Direction);
      proceed (EndPacket);

   state EndPacket:
      MyPort->wait (EMP, PReceived);

   state PReceived:
      Client->receive (ThePacket, ThePort);
      if (REQ_SET_PSR->put () == REJECTED)
          print ("\nSignal Rejected at QAR");
      proceed (WaitPacket);
};
```

A.9 Reassembly State Machine

```
RSM::perform {

   state BOMPktArr:
      BEtag = (*QABuffer)->COM_PDU_Hdr.BEtag;
      Rx_Seq_Num = ((*QABuffer)->DM_Hdr.Seq_Num + 1) % 16;
      Msg_Len = (*QABuffer)->DM_Trl.PayLoad_Len;
      if ((*QABuffer)->DM_Hdr.Seg_Type == SSM)
         proceed (Validation);
      proceed (NextPktArr);

   state NextPktArr:
      TheProcess->wait(SIGNAL, CheckPktStatus);

   state CheckPktStatus:
      if (Rx_Seq_Num != (*QABuffer)->DM_Hdr.Seq_Num) {
         print ("\n Sequence Number out of Sequence");
         proceed (Stop);
      }
      Msg_Len += (*QABuffer)->DM_Trl.PayLoad_Len;
      Rx_Seq_Num = (Rx_Seq_Num + 1) % 16;
      if (((*QABuffer)->DM_Hdr.Seg_Type == SSM) ||
                     ((*QABuffer)->DM_Hdr.Seg_Type == EOM))
         proceed (Validation);
      else
         proceed (NextPktArr);

   state Validation:               // Verified the Message recv
      Msg_Len -= (PHdrLngth * 2);  // Minus COMMON PDU Hdr & Trl
                                   //  8 bytes = 32 * 2
      if (Msg_Len != (*QABuffer)->COM_PDU_Trl.Length)
         print ("\n Message Length does not Match ");

      if (BEtag != (*QABuffer)->COM_PDU_Trl.BEtag)
         print ("\n BEtag does not Match ");
      proceed (Stop);

   state Stop:
      terminate ();
};
```

Appendix B

The Implementation of DQDB Observers

B.1 Slot Delay

```
#define  Dq ((Dqsm *)TheProcess)

observer SD_LR {
   long      see_request, n_slots[2], slot_delay[2];
   int       stat_id, pivot, CD_cntr;
   void      setup(int);
   void      resetLmt();
   states    {Resume, SetCntr, PriReq, PassingSlots, Verify};
   perform;
};

void SD_LR::setup (int id) {
   stat_id = id;
   see_request = (i*SegmentLength)/(long) TotalSlotTime;
   for (int i=0; i<2 ; i++) {
        n_slots[i] = 0;
        slot_delay[i] =0;
   }
   CD_cntr = pivot = 0;
};

void SD_LR::resetLmt() {
   pivot = Dq->Priority;
   CD_cntr = (*(Dq->CDM_cntr));
   if (pivot == 0)
```

```
        CD_cntr += (2+stat_id*2);
    if (pivot == 1)
        CD_cntr += (1+stat_id);
    n_slots[pivot] = 0;
    slot_delay[pivot] = CD_cntr+((see_request+1)*2);
};

SD_LR::perform {

    state Resume :
        inspect(stat_id, Dqsm, DqsmA, CountDown, SetCntr);
        inspect(stat_id, Dqsm, DqsmA, IncCntrs, PriReq);
        inspect(stat_id, Monitor, MtrA, SlotHeader, PassingSlots);
        inspect(stat_id, Transmitter, XtmA, EPacket, Verify);

    state SetCntr :
        resetLmt();
        proceed (Resume);

    state PriReq :
        if (Dq->Dqsm_State == COUNTDOWN)
            (slot_delay[Dq->Priority])++;
        proceed (Resume);

    state PassingSlots :
        (n_slots[0])++;
        (n_slots[1])++;
        (n_slots[2])++;
        proceed (Resume);

    state Verify :
        pivot = ((Transmitter *)TheProcess)->Priority+1;
        if (n_slots[pviot] > slot_delay[pivot]) {
            print (stat_id, "\nThe LR Observer at :  ");
            print ("\nSlot Delay exceed permitted limit\n");
        }
        proceed (Resume);
};
```

B.2 Queued Arbitrated Portion Observer

```
#define qap ((QAP *)TheProcess)
#define MCF ((MCF_Block *)TheProcess)->MCFBuffer
```

```
observer QAP_Obs {
   QASegment *BufPtr[6], *Buf;
   int      level;
   void      setup(int);
   states   {Resume, AccBus, Verify};
   perform;
};

void QAP_Obs::setup (int id) {
  stat_id = id;
  for (i = 0; i<6; i++)
       BufPtr[i] = NULL;
  level = 0;
};

QAP_Obs::perform {

   state Resume :
      inspect(stat_id, MCF_Block, ANY, WaitAccess, AccBus);
      inspect(stat_id, QAP, ANY, MovetoXtmBuf, Verify);

   state AccBus :
      if (MCF->Receiver >= MCF->Sender)
          BufPtr[MCF->TP] = MCF;
      else
          BufPtr[(MCF->TP)+3] = MCF;
      proceed (Resume);

   state Verify :
      if (qap->QABuf->VCI != ((Node *)TheStation)->VCI)
          print("\n Virtual Channel Identifier is incorrect");
      level=0;
      if (qap->Direction == LEFT)
          level=3;
      Buf = BufPtr[(qap->QABuf->TP)+level];
      BufPtr[(qap->QABuf->TP)+level] = NULL;
      if ((Buf->DM_Hdr.MID != qap->QABuf->DM_Hdr.MID) ||
         (Buf->DM_Hdr.Seg_Type != qap->QABuf->DM_Hdr.Seg_Type) ||
         (Buf->DM_Hdr.Seq_Num != qap->QABuf->DM_Hdr.Seq_Num))
          print ("\n Wrong QASegment at QAT ");
      proceed (Resume);
};
```

B.3 DQSM Observer

```
#define mtr  ((Monitor *)TheProcess)
#define Stn  ((Node *)TheStation)

observer Dqsm_Obs {
   int     REQ_cntr[6], CD_cntr[6], stat_id,
           level, st, Dq_State_0, Dq_State_1;
   void    setup(int);
   void    UpdateCntrs(int, int);
   void    IncCntrs(int, int, int);
   void    DecCntrs(int, int);
   states {Resume, ACFStatus, SelfReq, SetCD, Verify};
   perform;
};

void Dqsm_Obs::setup (int id) {
   stat_id = id;
   for (int j=0; j < 6; j++) {
        REQ_cntr[j] = 0;
        CD_cntr[j] = 0;
        }
   level = Dq_State_0 = Dq_State_1 = 0;
};
void Dqsm_Obs::UpdateCntrs (int lev, int Dq_St) {
   if (Dq_St == IDLE)
      (REQ_cntr[lev])++;
   else
      (CD_cntr[lev])++;
};
void Dqsm_Obs::IncCntrs (int lev, int Dq_0, int Dq_1) {
   if ((*(mtr->SlotPtr))->REQ_0)
       (REQ_cntr[lev])++;
   if ((*(mtr->SlotPtr))->REQ_1) {
       (REQ_cntr[1+lev])++;
        UpdateCntrs(lev, Dq_0);
   }
   if ((*(mtr->SlotPtr))->REQ_2) {
       (REQ_cntr[2+lev])++;
        UpdateCntrs(lev, Dq_0);
        UpdateCntrs((lev+1), Dq_1);
   }
};
void Dqsm_Obs::DecCntrs (int lev, int Dq_St) {
   if (Dq_St == IDLE) {
      if (REQ_cntr[lev] > 0)
         (REQ_cntr[lev])--;
   } else {
      if (CD_cntr[lev] > 0)
         (CD_cntr[lev])--;
   }
};
Dqsm_Obs::perform {

   state Resume :
      inspect(stat_id, Monitor, ANY, SlotHeader, ACFStatus);
      inspect(stat_id, QAP, ANY, MovetoXtmBuf, SelfReq);
      inspect(stat_id, Dqsm, ANY, CountDown, SetCD);
      inspect(stat_id, Dqsm, ANY, EmptySlot, Verify);

   state ACFStatus :
      if (mtr->Direction == LEFT) {
          if ((*(mtr->SlotPtr))->BUSY == NO) {
             DecCntrs(0, Stn->Dqsm_0_RL);
             DecCntrs(1, Stn->Dqsm_1_RL);
             DecCntrs(2, Stn->Dqsm_2_RL);
          }
          IncCntrs(3, Stn->Dqsm_0_LR, Stn->Dqsm_1_LR);
      } else {
          if ((*(mtr->SlotPtr))->BUSY == NO) {
             DecCntrs(3, Stn->Dqsm_0_LR);
             DecCntrs(4, Stn->Dqsm_1_LR);
             DecCntrs(5, Stn->Dqsm_2_LR);
          }
          IncCntrs(0, Stn->Dqsm_0_RL, Stn->Dqsm_1_RL);
      }
      proceed (Resume);

   state SelfReq :
      if (Qf->Direction == LEFT) {
          level = 0;
          Dq_State_0 = Stn->Dqsm_0_RL;
          Dq_State_1 = Stn->Dqsm_1_RL;
      } else {
          level = 3;
          Dq_State_0 = Stn->Dqsm_0_LR;
          Dq_State_1 = Stn->Dqsm_1_LR;
      }
      switch (Qf->Priority) {
        case 0 : break;
        case 1 : UpdateCntrs(level, Dq_State_0);
                 break;
        case 2 : UpdateCntrs(level, Dq_State_0);
                 UpdateCntrs((1+level), Dq_State_1);
                 break;
      }
      proceed (Resume);

   state SetCD :
      level = Dq->Priority;
      if (Dq->Direction == RIGHT)
          level+=3;
      CD_cntr[level] = REQ_cntr[level];
      REQ_cntr[level] = 0;
      proceed (Resume);

   state Verify :
      level = Dq->Priority;
      if (Dq->Direction == RIGHT)
          level+=3;
      if ((REQ_cntr[level]-(*(Dq->REQM_cntr))) > 1)
          print(stat_id,"\n Error --> REQM_cntr in Dqsm");
      if ((CD_cntr[level]-(*(Dq->CDM_cntr))) > 1)
          print(stat_id,"\n Error --> CD_cntr in Dqsm");
      proceed (Resume);
};
```

B.4 MCF Observer

```
#define McfB (((MCF_Block *)TheProcess)->MCFBuffer)

observer MCF_Obs {
   int      cntrs[3][4], MCP_Len, stat_id;
   void     setup(int);
   void     CheckStatus();
   void     UpDateTx(int, int);
   void     CheckBEtag(int, int, int);
   states   {Resume, FPacket, Verify};
   perform;
};
```

```
void MCF_Obs::setup (int id) {
  stat_id = id;
  for (j = 0; j < 3; j++)
    for (k = 0; k < 4; k++)
         cntrs[j][k] = 0;
  MCP_Len = 160;
};
void MCF_Obs::UpDateTx (int lev, int Tx_cntr) {
  cntrs[lev][1] += McfB->ILength;
  if (cntrs[lev][3] != Tx_cntr)
      print(lev, "\nTx Seq Cntr is out of order");
  cntrs[lev][3] = (cntrs[lev][3]+1)%16;
};
void MCF_Obs::CheckBEtag (int lev, int Msg_Len, int BEtag_cntr) {
  cntrs[lev][0] = Msg_Len;
  if (cntrs[lev][2] != BEtag_cntr)
      print(lev,"\n BEtag Cntr is incorrect");
};
void MCF_Obs::CheckStatus () {
  if (McfB->DM_Hdr.MID != (McfB->Sender+(McfB->TP*NumSt)))
      print ("\nMID is incorrect");
  switch (McfB->TP) {
      case 0 : UpDateTx(0, MCF->Tx_Seq_Num_0);
               break;
      case 1 : UpDateTx(1, MCF->Tx_Seq_Num_1);
               break;
      case 2 : UpDateTx(2, MCF->Tx_Seq_Num_2);
               break;
      } // switch
};

MCF_Obs::perform {

   state Resume :
      inspect(stat_id, MCF_Block, ANY, FDMPDU, FPacket);
      inspect(stat_id, MCF_Block, ANY, WaitAccess, Verify);

   state FPacket :
      switch (McfB->TP) {
         case 0 : CheckBEtag(0, Mcf->Msg_Len_0, Mcf->BEtag_cntr_0);
                  break;
         case 1 : CheckBEtag(1, Mcf->Msg_Len_1, Mcf->BEtag_cntr_1);
                  break;
         case 2 : CheckBEtag(2, Mcf->Msg_Len_2, Mcf->BEtag_cntr_2);
```

```
                  break;
      }
      proceed (Resume);

   state Verify :
      switch (McfB->DM_Hdr.Seg_Type) {
         case BOM,
              SSM : if (McfB->COM_PDU_Hdr.MCP_Hdr.DA != McfB->Receiver)
                        print ("\n Receiver Address is incorrect");
                    if (McfB->COM_PDU_Hdr.MCP_Hdr.SA != McfB->Sender)
                        print ("\n Sender Address is incorrect");
                    if (McfB->COM_PDU_Hdr.MCP_Hdr.PI != 1)
                        print ("\n Protocol Identifier is incorrect");
                    CheckStatus();
                    break;
         case COM : CheckStatus();
                    break;
         case EOM : CheckStatus();
                    if (cntrs[McfB->TP][2] != McfB->COM_PDU_Trl.BEtag)
                        print("\n Error --> Trl BEtag unmatch");
                    if (cntrs[McfB->TP][0] != cntrs[McfB->TP][1])
                        print("\n Error --> Message unequal length");
                    cntrs[McfB->TP][0] += MCP_Len;
                    if (cntrs[McfB->TP][0] != McfB->COM_PDU_Trl.Length)
                    cntrs[McfB->TP][2] = (cntrs[McfB->TP][2]+1)%256;
                    cntrs[McfB->TP][0] = cntrs[McfB->TP][1] = 0;
                    break;
      } // switch
      proceed (Resume);
};
```

B.5 RQSM Observer

```
#define Req  ((Rqsm *)TheProcess)

observer RQSM_Obs {
   int    Req_Q[6], ReqBit[6], stat_id, level;
   void   setup(int);
   void   SetLeveL(int, int, int, int *);
   states {Resume, IncCntr, CheckCntr, CheckReq, Verify};
   perform;
};
```

```
void RQSM_Obs::setup (int id) {
   stat_id = id;
   for (j=0; j < 6; j++) {
        Req_Q[j]  = 0;
        ReqBit[j] = YES;
   }
   level = 0;
};
void RQSM_Obs::SetLeveL (int Dir, int Pri, int Side, int lev) {
    lev = Pri;
    if (Dir == Side)
        lev+=3;
};
RQSM_Obs::perform {

   state Resume :
      inspect(stat_id, Dqsm, ANY, CountDown, IncCntr);
      inspect(stat_id, Rqsm, ANY, LocalReq, CheckCntr);
      inspect(stat_id, Monitor,  ANY, SlotHeader, CheckReq);
      inspect(stat_id, Rqsm, ANY, SetReqBit, Verify);

   state IncCntr :
      SetLeveL(Dq->Direction, Dq->Priority, LEFT, level);
      (Req_Q[level])++;
      proceed (Resume);

   state CompareREQ :
      SetLeveL(Req->Direction, Req->Priority, RIGHT, &level);
      if ( Req_Q[level] != (*(Req->REQM_Q_cntr)) )
          print(stat_id,"\nREQM_Q_cntr in STATE LocalReq at : ");
      proceed (Resume);

   state ReqBitStatus :
      SetLeveL(mtr->Direction, 0, RIGHT, level);
      if ((*(mtr->SlotPtr))->REQ_0 == NO)
         ReqBit[level+0] = NO;
      if ((*(mtr->SlotPtr))->REQ_1 == NO)
         ReqBit[level+1] = NO;
      if ((*(mtr->SlotPtr))->REQ_2 == NO)
         ReqBit[level+2] = NO;
      proceed (Resume);

   state Verify :
      SetLeveL(Req->Direction, Req->Priority, RIGHT, level);
```

```
        (Req_Q[level])--;
        if (Req_Q[level] != (*(Req->REQM_Q_cntr)))
            print(stat_id,"\nREQM_Q_cntr in STATE SetReqBit at : ");
        if (ReqBit[level] != NO) {
            print("\n Error --> ReqBit has been set");
            ReqBit[level] = YES;
        }
        proceed (Resume);
};
```

B.6 Queued Arbitrated Receive Observer

```
#define Qarb  (*((QAF_Recv_Block *)TheProcess)->QABuffer)
#define sg  ((QASegment *)ThePacket)

observer QAR_Obs {
   QASegment *BufPtr[NStations*3], *Buf;
   int       MList[NStations*3], stat_id, MID;
   void       setup(int);
   states    {Resume, BusySlot, PktArr, FPacket, Verify};
   perform;
};
void QAR_Obs::setup (int id) {
  stat_id = id;
  for (j = 0; j<(NStations*3); j++) {
       BufPtr[j] = NULL;
       MList[j] = -1;
  }
};

QAR_Obs::perform {

   state Resume :
      inspect(stat_id, Monitor, ANY, SlotBody, BusySlot );
      inspect(stat_id, QAR, ANY, CheckPktStatus, PktArr);
      inspect(stat_id, QAR, ANY, BOMPktArr, FPacket);
      inspect(stat_id, QAR, ANY, EndPacket, Verify);

   state BusySlot :
      MID = sg->DM_Hdr.MID;
      BufPtr[MID] = sg;
      if ((sg->DM_Hdr.Seg_Type == BOM) &&
          (sg->COM_PDU_Hdr.MCP_Hdr.DA == stat_id) &&
            ((Stn->MIDList[MID] != MID))
              MList[MID] = MID;
        proceed (Resume);

    state PktArr :
        MID = Qarb->DM_Hdr.MID;
        Buf = BurPtr[MID];
        if ((Buf->VCI != Qarb->VCI) ||
            (Buf->DM_Hdr.MID != Qarb->DM_Hdr.MID) ||
            (Buf->DM_Hdr.Seg_Type != Qarb->DM_Hdr.Seg_Type) )
             print ("\n Error --> QASeg at QAR in STATE CheckStatus");
        if ((MList[MID] == MID) && (Buf->DM_Hdr.Seg_Type == EOM) )
             MList[MID]=-1;
        proceed (Resume);

    state FPacket :
        MID = Qarb->DM_Hdr.MID;
        if (MList[MID] != Stn->MIDList[MID])
            print ("\n Error --> BOM at QAR STATE FPacket");
        proceed (Resume);

    state Verify :
        if (Qarb->Receiver != TheStation->getId () )
           print ("\n Error --> in QAR at STATE EndPacket, wrong DA");
        proceed (Resume);
};
```

B.7 Transmitter Observer

```
#define Xtm ((Transmitter *)TheProcess)

observer Xmitter_Obs {
    enum   {NON, AFT};
    int    stat_id, level, DqStatus[6];
    void   setup (int);
    states {Resume, EnterCD, Verify};
    perform;
};

void Xmitter_Obs::setup (int id) {
    stat_id = id;
    for (int j=0; j<6; j++)
         DqStatus[j] = NON;
```

```
   level = 0;
};

Xmitter_Obs::perform {

   state Resume :
      inspect(stat_id, Dqsm, ANY, CountDown, EnterCD);
      inspect(stat_id, Transmitter, ANY, WaitTransmit, Verify);

   state EnterCD :
      level = Dq->Priority;
      if (Dq->Direction == RIGHT)
          level+=3;
      DqStatus[level] = AFT;
      proceed (Resume);

   state Verify :
      level = 0;
      if (Xtm->QABuffer->TP != Xtm->Priority)
          print("\n QASegment in wrong Xtm buffer.");
      if (Xtm->Direction == RIGHT) {
          if (Xtm->QABuffer->Sender >= Xtm->QABuffer->Receiver)
              print("\n QASegment should be Xtm on Bus B");
          level += 3;
      } else {
          if (Xtm->QABuffer->Sender < Xtm->QABuffer->Receiver)
              print("\n QASegment should be Xtm on Bus A");
      }
      switch (Xtm->Priority) {
        case 0 : if (DqStatus[level+2] == AFT)
                     print("\n Error --> Xtm Pri 0 > Pri 2");
                 if (DqStatus[level+1] == AFT)
                     print("\n Error --> Xtm Pri 0 > Pri 1");
                 DqStatus[level] = NON;
                 break;
        case 1 : if (DqStatus[level+2] == AFT)
                     print("\n Error --> Xtm Pri 1 > Pri 2");
                 DqStatus[level+1] = NON;
                 break;
        case 2 : DqStatus[level+2] = NON;
                 break;
      }
      proceed (Resume);
};
```